Jani Hietala

Real-time two-way data transfer with a Digital Twin via web interface

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 05.02.2020
Supervisor: Prof. Kari Tammi
Advisor: M.Sc. (Tech.) Heikki Laaki

Aalto University
School of Engineering

Aalto University, P.O. Box 11000, 00076 Aalto
www.aalto.fi

Abstract of master's thesis

Author Jani Hietala

Title of thesis Real-time two-way data transfer with a Digital Twin via web interface

Master programme Mechanical Engineering **Code** ENG25

Thesis supervisor Prof. Kari Tammi

Thesis advisor M.Sc. (Tech.) Heikki Laaki

Date 05.02.2020 **Number of pages** 48+6 **Language** English

Abstract

Technological advancements in industry have paved way for the fourth industrial revolution called Industry 4.0. One critical aspect of this revolution is the usage of digital twins in product design, production, and service. A common depiction of a digital twin consists of three parts: a physical twin, its digital twin, and the data exchanged between them. In industry, one common solution for the data exchange between the digital twin and its physical twin is OPC Unified Architecture (OPC UA) communication protocol. The protocol provides a solution to collect data from devices along an entire production line. Communication with OPC UA servers requires carefully studying the protocol specification, which can deter new developers from creating applications with the data collected by the servers.

The goal of this thesis is to develop a web-based application program interface (API) that simplifies transferring data with an OPC UA server. The API is intended to be made with a popular technology that is already widely known among developers. It would lower the learning curve for utilizing the data on OPC UA servers. Thus, more developers can be tempted to start developing applications with the data. As the API is web-based, it is accessible by any web capable device bringing the data available to virtually any programming language and platform.

Requirements for the API beyond the functionalities concern its efficiency and capability of handling real-time data exchange situations. To test the API's performance, a case study is made: a web-based control application. The control application uses the API in real-time to both write control signals to and read sensor values from the OPC UA server. The API performance is evaluated by measuring its request completion time in both controlled environment and real use cases.

The developed API was considered to be fast enough for user-based input and even applications that required fast synchronisation of values from different data sources. However, the API did add considerable latency compared using the OPC UA server directly which might be a problem in some applications that require extremely time sensitive data from the data server.

Keywords Digital twin, IoT, OPC Unified Architecture, GraphQL, application programming interface

Tekijä Jani Hietala

Työn nimi Reaaliaikainen kaksisuuntainen tiedonsiirto digitaalisen kaksosen kanssa verkkorajapinnan kautta

Maisteriohjelma Konetekniikka **Koodi** ENG25

Työn valvoja Prof. Kari Tammi

Työn ohjaaja DI Heikki Laaki

Päivämäärä 05.02.2020 **Sivumäärä** 48+6 **Kieli** Englanti

Tiivistelmä

Teknologian kehitys teollisuudessa on mahdollistanut neljännen teollisuuden vallankumouksen, jota kutsutaan myös nimellä Industry 4.0. Yksi tämän vallankumouksen tärkeitä puolia on digitaalisten kaksosten hyödyntäminen tuotesuunnittelussa, tuotannossa, sekä huoltamisessa. Yleensä digitaalisen kaksosen käsitteen sanotaan koostuvan kolmesta osasta: fyysisestä kaksosesta, sen digitaalisesta kaksosesta ja niiden välisestä tiedonsiirrosta. Teollisuudessa eräs yleinen tapa toteuttaa tiedonsiirto fyysisen ja digitaalisen kaksosen välillä on OPC Unified Architecture (OPC UA) -kommunikaatioprotokollalla. Protokolla tarjoaa ratkaisun datan keruuseen koko tuotantolinjan prosesseista. Kommunikointi OPC UA -palvelimen kanssa vaatii protokollan määrittelyyn syventymistä, mikä saattaa lannistaa uusia kehittäjiä, jotka voisivat luoda sovelluksia palvelimen keräämälle datalle.

Työn tavoitteena on kehittää web-pohjainen ohjelmointirajapinta, joka yksinkertaistaa sovellusten kehittämistä OPC UA -palvelimen kanssa. Ohjelmointirajapinta on tarkoitus kehittää ohjelmistokehittäjille tunnetulla teknologialla. Web-rajapinnalla voi laskea oppimiskynnystä OPC UA -palvelimen datan hyödyntämiseen. Tällöin uudet kehittäjät saattavat olla kiinnostuneempia kehittämään käyttökohteita datalle. Web-rajapinnalla on mahdollista tarjota data mille tahansa alustalle tai ohjelmointikielelle, jolla on web-ominaisuudet.

Toiminnallisuuksien lisäksi rajapinnan vaatimukset kohdistuvat sen tehokkuuteen ja kykyyn käsitellä reaaliaikaista tiedonsiirtoa. Rajapinnan testausta varten tehdään tapaustutkimus: Web-pohjainen ohjaussovellus. Ohjaussovellus käyttää rajapintaa reaaliajassa kirjoittamaan ohjaussignaaleja, sekä lukemaan anturiarvoja OPC UA -palvelimelta. Rajapinnan suorituskykyä arvioidaan mittaamalla sen pyyntöjen suoritusaikoja sekä hallitussa ympäristössä, että tosikäyttötilanteissa.

Kehitetty ohjelmointirajapinta todetaan testeissä tarpeeksi nopeaksi käyttäjäsyötteelle, sekä ohjelmille, jotka tarvitsevat nopeaa synkronointia arvoille eri tietolähteistä. Rajapinta kuitenkin lisäsi merkittävän viiveen verratessa sitä OPC UA -palvelimen suoraan käyttöön. Tämä saattaa aiheuttaa ongelmia joissain käyttötilanteissa, jotka tarvitsevat tarkkaa aikariippuvaista dataa palvelimelta.

Avainsanat Digitaalinen kaksonen, IoT, OPC Unified Architecture, GraphQL, ohjelmointi-rajapinta

Preface

I would like to thank my thesis advisor Heikki Laaki for his valuable input and guidance on the writing process, and for providing me with the thesis topic. Thanks also to my supervisor Professor Kari Tammi for his input and help. Thanks to Juuso Autiosalo and Riku Ala-Laurinaho, who gave me insight on the concept of digital twin and created a friendly working environment. I thank the DigiTwin consortium for giving me an opportunity be part of an interesting project.

Finally, thanks to my friends, family and girlfriend who supported me through the writing of this thesis and my studies.

Espoo 5.2.2020

Jani Hietala

Contents

Abbreviations

API	Application Program Interface
DRM	Digital Rights Management
GraphQL	Graph Query Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IoT	Internet of Things
OPC	Open Platform Communications
OPC AE	OPC Alarms & Events
OPC DA	OPC Data Access
OPC DX	OPC Data eXchange
OPC HAD	OPC Historical Data
OPC UA	OPC Unified Architecture
OPC XML DA	OPC XML Data Access
PLC	Programmable Logic Controller
PLM	Product Lifecycle Management
REST	Representational State Transfer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language

1 Introduction

Technological advancements have paved way for the fourth industrial revolution called Industry 4.0. This revolution is expected to occur due to increasing capabilities for automated manufacturing processes, data collection, and internet access on the plant floor. One of the increasingly popular concepts is called a digital twin. A project group of researchers at the Aalto Industrial Internet Campus (AIIC) are studying the digital twin concept and its potential. As part of the research an industrial overhead crane called Ilmatar was installed to one of the AIIC laboratories. Ilmatar crane has capabilities beyond the common industrial cranes due to multitude of sensors and a programmable logic controller (PLC). The project group is interested in developing a digital twin for the crane. Thus, this thesis project is also looking to further develop the Ilmatar crane's digital twin.

The motivation for this thesis' work can be found from the increasing usage of digital twins in industry. Digital twins require large amounts of data to be collected from the products in various parts of their manufacturing and utilization. A common implementation of the data collection is done by building data servers following Open Platform Communications Unified Architecture (OPC UA) specifications. The OPC UA specifications instruct how to build servers that collect and serve data as parts of systems. In addition to building an OPC UA server, communicating with one requires studying the aforementioned specifications. This can be time-consuming to new developers, which is not preferred when the focus is on creating applications with the available data. For this problem, a well-known Application Program Interface (API) could provide a solution. A popular API could bring in new interested developers compared to having the developers first study the OPC UA specifications.

1.1 Objective

The DigiTwin project group builds and develops a digital twin for an overhead crane. Researchers commonly describe a digital twin to have three key characteristics: the physical twin, the digital twin, and the data flow between them (Tao and Zhang, 2017). This thesis aims to bring a solution to the two-way communication between the physical twin and its digital twin by enabling programmatically effortless data transfer. This is due to using a commonly used web interface as framework which is accessible from any platform with web capabilities. The web interface enables developers to easily create new applications to be integrated as part of the crane's digital twin.

The more specific objective for this thesis is to build an API, which provides applications an access to the live sensor data and controls on the Ilmatar crane server. The API aggregates multiple data servers and presents their information models seamlessly to any clients accessing the API. There are several key requirements for the API to fulfil. The API should be developer friendly, provide potentially real-time communication with the server, be accessible from any HTTP (Hypertext Transfer Protocol) capable device, and aggregate multiple different servers. This work tries to find an answer to the question of *how to improve an OPC UA server for more friendly consumption of data by common developers*. Another question to which an answer is searched for is *how much latency can the resulted API cause while still being usable in most real-time and user applications*. The resulted API's usability is expected to be considerably better if an existing, well known and already proved API technology is used. The latency caused by the API needs to be measured and tested in some way for comparable results.

The resulted API was tested by a case study: A web-based crane control and monitor application that both reads and writes data to the digital twin via the API. Functionally, the application provides monitoring data from the Ilmatar crane server and gives access to the crane's controls. The application is web based and is used as a decentralized control platform with capabilities for hosting and executing various control and monitoring schemes. With the control application, the API's applicability to different applications can be evaluated via usage tests.

1.2 Scope

This thesis focuses on building a developer friendly and expandable API for aggregating OPC UA servers as part of the Ilmatar crane's digital twin. The thesis does not cover in depth the concept of a digital twin. This is because digital twin is a broad subject and most of it is irrelevant for this thesis's work. Instead, this thesis will try to present the benefits and methods of building a developer friendly API for OPC UA servers. The API is not intended to completely expose the full information models of OPC UA server nodes as only part of the information serves towards this project's objectives. Latency measurements are made to compare the GraphQL API's performance to directly using the OPC UA server via its built in API. This is also supported by a case study for which a web-based control and monitoring application is built to highlight the digital twin of Ilmatar crane.

1.3 Structure of the work

This thesis will first introduce the digital twin concept to a sufficient level to understand this projects relevance to it. Secondly this work will discuss an OPC UA server's architecture and its importance as a common open source standard in industrial 4.0 systems. In this thesis a GraphQL API is developed for an OPC UA server to have its data more accessible when developing future applications. Benefits of the GraphQL query language for this project are discussed and compared to RESTful APIs. The GraphQL API's feasibility for real-time applications is measured through query execution times. The results are compared to communicating with the OPC UA server directly. For a case study a web-based control and monitoring application that utilizes the GraphQL API is developed. Its performance in real use cases and controlled environment is measured through tests. Finally, discussion is made on how the built GraphQL API performs and compares to existing similar APIs in literature. GraphQL API's capability is discussed as a possible solution for real-time applications when comparing it to directly using the OPC UA server.

1.4 Digital twin

The term digital twin could be interpreted as any kind of a digitalized twin of some object, be it a physical device, virtual device or even a human. The definition of a digital twin is not always clear in literature (Autiosalo *et al.*, 2019). Companies and researchers may use the term as suits them best or how they interpret it. However, over time this term has seen the most use in describing digitalized models of physical objects or systems (Glaessgen and Stargel, 2012; Grieves and Vickers, 2016; Mukherjee and DebRoy, 2019). Data would be collected from these physical objects and systems and then be modelled and simulated into digital models which would try to mirror their physical counterpart. As technology advances these digital twins can become more and more accurate representations of their physical twins. With higher digital twin accuracy, more useful applications can also be potentially developed for the digital twins.

Nowadays the concept of a digital twin has become more defined. Most commonly the term digital twin is used to define a digital model or simulation of a system that is high-fidelity and an accurate real-time representation of its physical twin. A common definition comes from Glaessgen and Stargel (2012): "The Digital Twin is an integrated multiphysics, multiscale, probabilistic simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror life of its corresponding flying twin." (Glaessgen and Stargel, 2012). Scientific community has commonly recognized this as the definition of a digital twin (Tao and Zhang, 2017). As stated before, the term "digital twin" is vague as itself and can be applied to many different fields of industries and use cases.

Tao and Zhang (2017) have expanded on the above definition of a digital twin a bit further by saying that the digital twin requires three parts to function. Firstly, it needs the physical twin that represents any object or system from which the necessary data is collected. Secondly, a digital twin that mirrors its physical twin. And lastly, the connected data that should flow freely between the twins. The digital twin is continuously synchronised to represent the physical twin's state with the connected data. The physical twin's state can also be updated from changes to the digital twin, but this depends on the use case for the twin.

An example use case for a digital twin was proposed by Söderberg *et al* (2017) where a digital twin was used in real time geometry assurance to present how the industry could move from mass production to more individualized production (Söderberg *et al.*, 2017). Another use case was presented by Mukherjee and DebRoy (2019) where they explained how a digital twin of a 3D printing machine could be used to reduce trial and error in 3D printing while also reducing defects and shortening time from design to production (Mukherjee and DebRoy, 2019). Finally, there is a use case by Bielefeldt, Hochhalter and Hartl (2015) in which a digital twin model was built to monitor the performance of aircraft subsystems and detect fatigue cracks in the aircraft's structure (Bielefeldt, Hochhalter and Hartl, 2015).

In above examples the usage of a digital twin seems to commonly focus on assisting product development, manufacturing and maintenance. Thus, most of the use cases can be said to relate to Product Lifecycle Management (PLM). PLM is managing a product throughout its concept phase all the way to its disposal with the main objective of increasing product revenues, reducing costs, and maximising value of current and future products (Lämmer and Theiss, 2015). PLM introduces numerous difficult challenges to which Tao *et al* (2018) proposed a solution using a digital twin (Tao *et al.*, 2018). Particularly, the paper emphasized digital twin's potential applications in product's design, manufacturing and service. Figure 1 illustrates what data can be collected to a digital twin from every part of a product's lifecycle. The produced data can then be used to simulate and estimate what changes to the product affect its quality and in which ways. With the information received from simulations and estimations, future generations of the product can be improved.

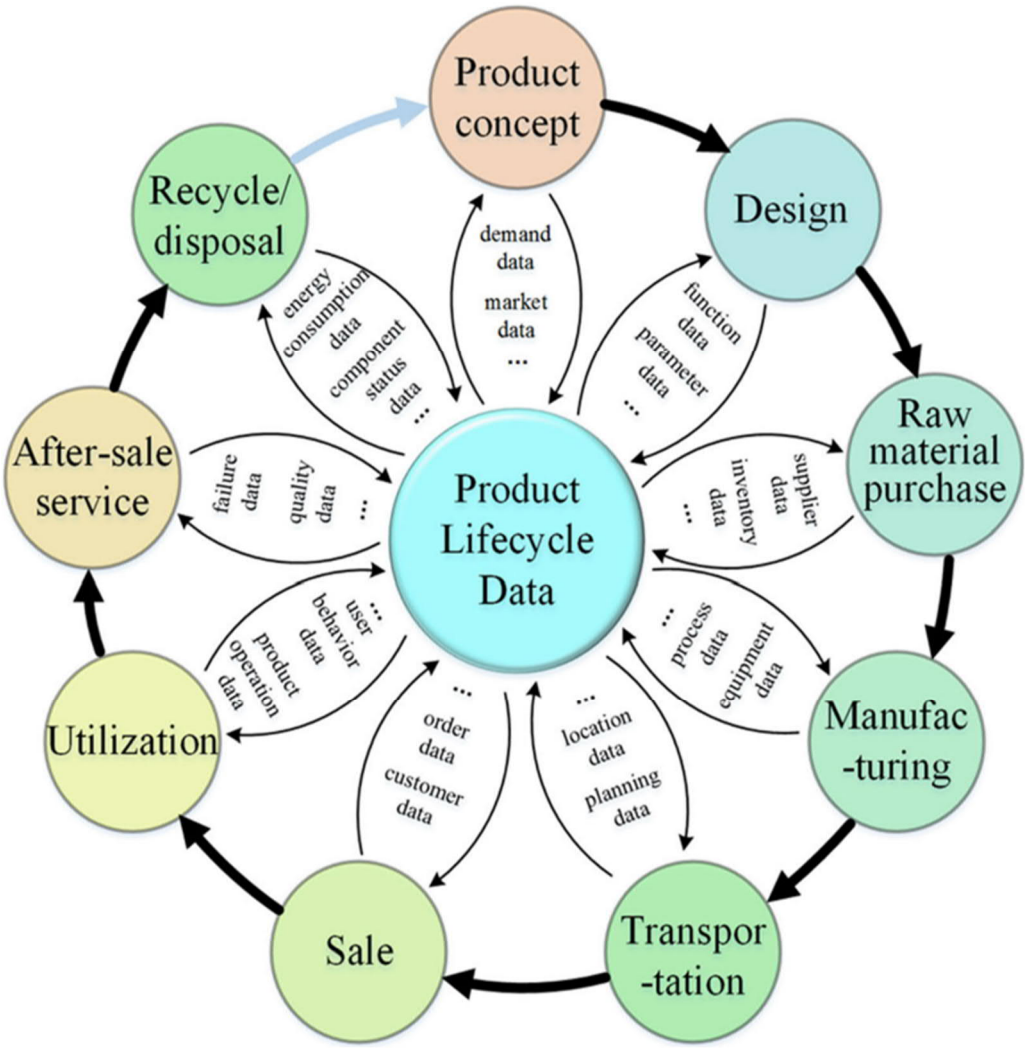

Figure 1. Product lifecycle and related data for a digital twin (Tao *et al.*, 2018).

In an ideal scenario for a fleet of products, each single product would have its own digital twin that has data from each part of its lifecycle. Each of these products would end up living its own life in differing usage conditions and maintenance requirements. The collected data could be used to better customize future products for different environments and applications. Any variations in design or manufacturing to a single product can be observed to affect the data collected during its utilization. By properly managing the design and manufacturing parameters of the product, at the end of the product generation, the products manufactured would be of the highest possible quality. In practice this would mean, for example, getting rid of unnecessary safety factoring, savings in material costs, and product would survive through its planned lifespan at high accuracy. Digital twin simulation based on the collected data could also be used to predict the effects of events that can have an impact on the product operation. Slowly incurring damage could be mitigated by changing the product operation as necessary. Simulations can also predict how the unexpected usage can affect the product's life expectancy.

1.4.1 Relevance

In their paper, Autiosalo *et al.* (2019) have proposed a list of features that can be used to distinguish different parts of digital twins (Autiosalo *et al.*, 2019). Figure 2 illustrates these parts by dividing them by their differences in functionalities. These functionalities can exist as part of any digital twin. A part of the Ilmatar's digital twin is created with various sensors on the crane and an OPC UA server that collects and views the sensor data in a unified data structure. In Figure 2, the OPC UA server can be depicted as a data link between the crane PLC (i.e. coupling in Figure 2) and any other parts of the digital twin. While the OPC UA server does not need to be the only data link between parts of the digital twin, it is currently the central data link in the digital twin built for the Ilmatar crane. Thus, in Ilmatar's case, the digital twin data link at low level is formed by the OPC UA server.

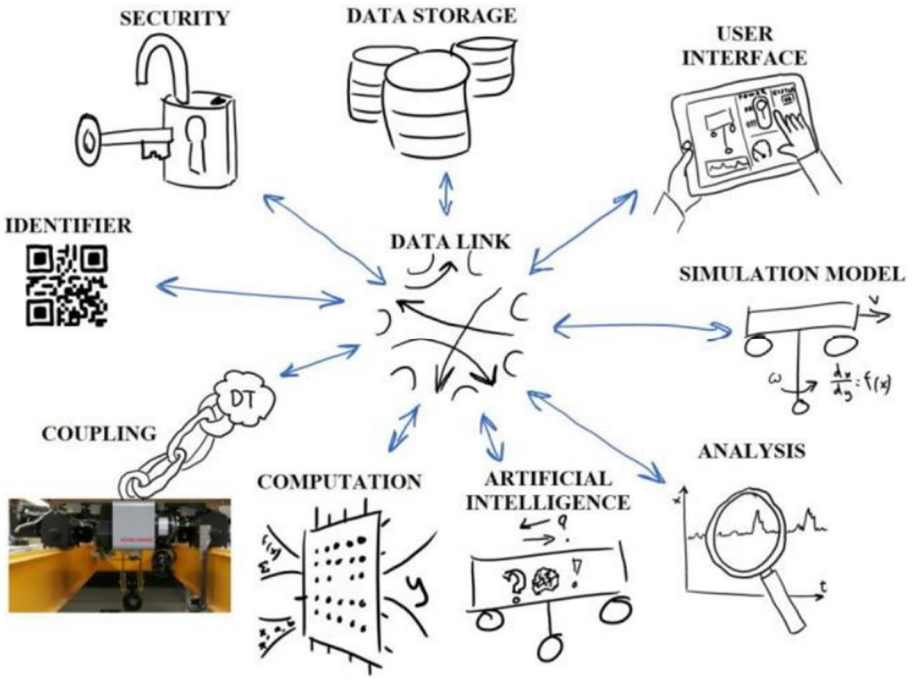

Figure 2. Concept of a digital twin's potential features (Autiosalo *et al.*, 2019).

The OPC UA server provides real time data of the Ilmatar crane status and can provide it forward to other functions of the digital twin. It is also possible to send commands via the OPC UA server to control the Ilmatar crane. The accessibility of this OPC UA server data is the main purpose of this thesis and is important for any applications that intend to use the crane's digital twin. Currently, real time data and control of the Ilmatar crane can be achieved using the OPC UA communication protocols. However, this requires studying the OPC UA specifications to great extent which can be time consuming for new developers. For accessibility, a common, well documented and developer friendly API in this case is beneficial. This API should also be able to fulfil the requirements of a digital twin such as real-time two-way data transfer at acceptable level. Furthermore, the API will be used as a data link between the crane controls and a user interface built as a case study. The user interface is used to monitor data and control the crane.

Some research to digital twins has been done before with concepts a bit similar to this thesis' work. In their research, Laaki *et al*. (2019) have built a digital twin for real-time remote control over mobile network (Laaki, Miche and Tammi, 2019). The built prototype system consisted of a virtual reality environment and a robotic arm, which communicated via a

digital twin over mobile networks. The user could remotely control the robotic arm's digital twin in the virtual world, and the robotic arm would move to match its digital twin. The system ran in real-time which is also one goal for this thesis' web API as the datalink between the Ilmatar crane and the case study's user interface. In the paper, time sensitive controls were sent to the robotic arm using User Datagram Protocol (UDP) which is commonly used in real-time applications due to its low latency. Despite this, the web API is planned to use Hypertext Transfer Protocol (HTTP) as application layer protocol which uses Transmission Control Protocol (TCP) for the transport layer. TCP ensures that the data packets are successfully transferred over the network but at higher latency (Kurose and Ross, 2013). More on HTTP and TCP can be found in the HTTP section. Reason for using HTTP stems from the web API being planned to function alike to a web data server. Real-time data transfer is second priority to usability and accessibility via web. TCP is optimal when the API is required to reliably return large amounts of data from the OPC UA server (Kurose and Ross, 2013).

Security is also important to consider when building digital twins that are potentially accessed from the public network. In the paper of Laaki *et al*. (2019), security needs were analysed because the application for the digital twin was in remote surgery. For legal reasons, intellectual property may need to be protected as well as the data contained in the digital twin. Even more importantly, any remote control over web should be secure so that no attack to the system can put people or property at risk. These vulnerabilities could be prevented by measures such as using Digital Rights Management (DRM), binding the digital twin to specific hardware, and ensuring strong user authentication (Laaki, Miche and Tammi, 2019). The developed web API and user interface in this work do not take into account many of these security concerns. This is due to the API being developed for internal use, and Ilmatar already has security measures to prevent unsupervised or unauthorised control of the crane. If the web API or user interface is later expanded for wider use, these security concerns should be accounted for.

2 Methods

The methods section introduces in depth the relevant technologies that were a crucial part of this thesis work, and what was developed using the technologies. Based on the literature introduced and summarized in this section, a GraphQL API web interface was developed. The GraphQL API is the main component developed as part of this thesis. In the GraphQL API section, the GraphQL API's function, requirements, and operation are introduced. As a case study, a web-based control application for the Ilmatar crane is developed. In the case study section, the control application is introduced and its function in as part of the Ilmatar digital twin is discussed.

2.1 OPC Unified Architecture

In this section, the OPC Unified Architecture (OPC UA) is introduced and its main functionalities are presented. The basic OPC UA server, its functionalities and communication are summarized to the extent that was deemed necessary for this thesis' project. Finally, the Ilmatar crane's OPC UA server is shortly introduced. How the OPC UA server specification is applied to the server is also discussed.

2.1.1 Introduction

Open Platform Communications (OPC) is an interoperability standard for exchange of data in an industrial space (OPC Foundation, 2015). OPC standard started in 1996 with OPC Classic. Its purpose was to abstract Programmable Logic Controller (PLC) specific protocols into a standardized interface. It was initially restricted to only Windows operating system. Today, the OPC standard is platform independent with an objective to ensure data communication between devices from multiple different vendors. OPC has become a series of specifications to define an interface between clients and servers as well as servers and servers. These specifications are being developed by industry vendors, end-users and software developers. OPC Foundation maintains and develops the OPC standard.

OPC Classic protocols are independent of each other and can be divided to five parts (Harmo, 2014). Each of these parts have their own write and read services, and have no common functionalities. Data Access (DA) is a data sharing protocol. It enables real time data to be shared among and from control and automation systems. Alarms & Events (AE) is a specification for data subscription. This means that clients can get notifications for any events or changes in values. Historical Data (HAD) is a specification for accessing past values. DA and AE do not do any data storing and thus retrieving past values was not possible before HAD. The last two specifications of OPC Classic are XML Data Access (XML DA), which brings Extensible Markup Language (XML) support to OPC, and Data eXchange (DX), which is similar to DA but focuses on "horizontal" data exchange between devices in a system (Matrikon, 2019).

OPC Unified Architecture was later developed to unify the specifications of OPC Classic. Reasons for the development were security and data modelling challenges that had to be addressed with the introduction of service-oriented architectures in manufacturing systems (OPC Foundation, 2015). On OPC UA, due to object structured information models, all data can be found in a unified address space instead of having the data separated like in OPC Classic. OPC UA thus supports features such as historical events, multiple hierarchies and providing methods and programs because all data is accessible and related to each other (Leitner and Mahnke, 2006). According to the OPC Foundation the OPC UA should provide

a scalable, future-proof and extensible open-platform architecture. The biggest difference between OPC UA and OPC Classic is the fact that OPC UA is platform independent, unlike the OPC Classic which heavily depended on OLE and DCOM technologies developed by Microsoft (Harmo, 2014).

Figure 3 illustrates an example of an OPC UA server and its data flow when operating with clients and real objects. The real objects in the figure would be physical or software objects that the server has access to. The server application is the code that implements the function of the server. Nodes are commonly representations of the real objects and are accessible by client service requests. The AddressSpace is the set of Nodes inside the server. It can be split into Views that restrict what Nodes the server shows to clients. Subscriptions are used to create monitored items that track changes to Node values and notify the client if certain thresholds are met. (OPC Foundation, 2017a)

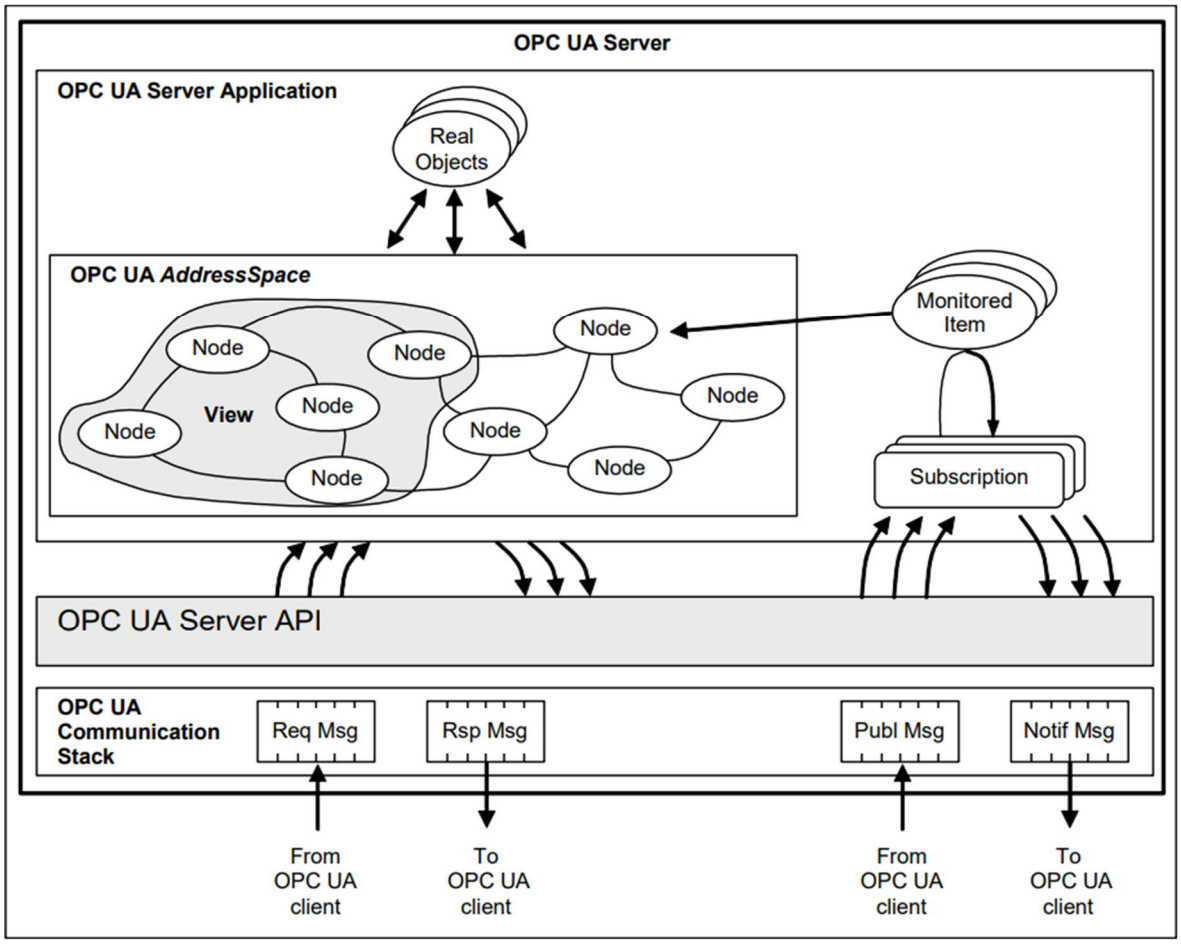

Figure 3. OPC UA Server architecture (OPC Foundation, 2017a).

Communication between the client and the server can be done with several different communication protocols as specified in the OPC UA specification. This leaves the decision of the protocol used to the end users instead of the OPC vendor. The communication protocols that are defined in the specifications are OPC UA TCP (Transmission Control Protocol), HTTPS and WebSockets. From these the users can make the decision based on the protocol performance and compatibility with their applications. (OPC Foundation, 2017d)

The importance of a standard for plant floor communications has risen since the industrial revolution Industry 4.0. As the number of connected components has increased, OPC UA standard has been able to answer some problems of this communication between these numerous subsystems. OPC UA has a long history of development, openly available specifications and wide library support for multiple programming languages. These features make the OPC UA architecture an attractive choice of for companies wanting to modernize manufacturing processes. Thus, the OPC UA has become a key part in practical implementations of digital twins in manufacturing industries.

2.1.2 Address Space

AddressSpace is a set of *Nodes* inside the OPC UA *Server* (OPC Foundation, 2017b). OPC UA *Servers* use the *AddressSpace* to represent *Nodes* to *Clients* in a standard way. *Nodes* are objects that are used to store and represent any information. Figure 4 illustrates the information stored in a *Node Model*. It consists of attributes that characterise the *Node* and references that define relationships to other *Nodes*. This section introduces some core concepts of OPC UA *AddressSpace* that are useful to know to understand some decisions made in this thesis' work. The explanations are quite brief and thus it is also recommended to familiarize yourself with the OPC UA specifications on topics relevant to your work.

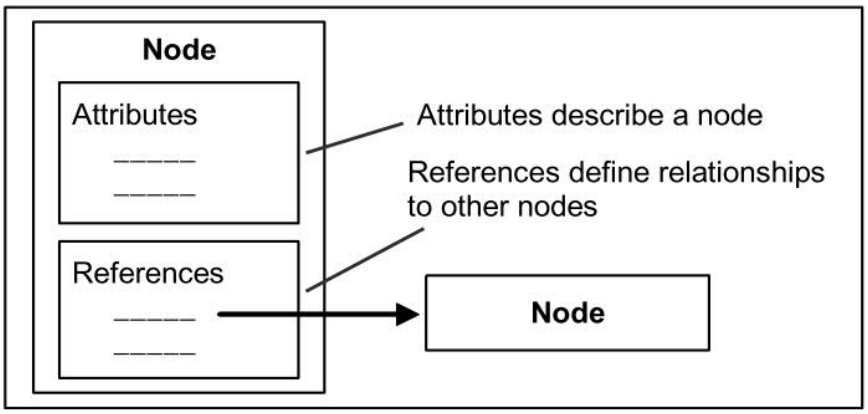

Figure 4. AddressSpace Node Model (OPC Foundation, 2017b).

The *Node Objects* belong to one of many *NodeClasses* that define the function, or more accurately, attributes of the *Node*. *NodeClasses* are standard and not meant to be modified or created by the *Client* or the *Server*. Table 1 presents the *NodeClasses* and their functional purposes. The *Base NodeClass* defines the attributes that all *NodeClasses* inherit. These attributes include information such as *NodeId*, which is the unique identifier used to find the *Node* in the *AddressSpace*, or *DisplayName*, which is the name to be displayed to users viewing the node.

Table 1. OPC UA NodeClasses and their descriptions (OPC Foundation, 2017b).

NodeClass	*Description*
Base	Base NodeClass from which all other classes are derived. Defines the base Attributes that all other NodeClasses inherit.
ReferenceType	Definitions for ReferenceType Nodes. References from one Node to another are defined as ReferenceType Nodes.
View	Defines a customized subset of Nodes. This enables viewing only the desired nodes from the AddressSpace. Nodes contained in a View class are all accessible by browsing references starting from the View Node.
Object	"Represents systems, system components, real-world objects and software objects." (OPC Foundation, 2017b)
ObjectType	Definitions for Objects. ObjectType FolderType has the sole purpose of organizing the AddressSpace. FolderType should only be referenced to by a View or another FolderType.
Variable	Variables are Properties or DataVariables of Nodes. Properties define characteristics of a Node. DataVariables are defined with a Variable NodeClass and represent content of an Object. Represents a value defined by VariableTypes.
VariableType	Definitions for Variables.
Method	Definitions for callable functions.
DataType	Describes the syntax of a Variable value. Variables and VariableTypes point to a DataType with their DataType Attribute.

The *NodeClass* of a *Node* defines what attributes the *Node* has. Attributes define metadata for *Nodes* that are instantiated from *NodeClasses* for all *Nodes*. The *Base NodeClass* defines in total ten attributes that should be available for any *Node*. Four of those are mandatory and the rest optional. The mandatory attributes describe the identity and necessary information to separate it from other *Nodes* in the *AddressSpace*. The optional attributes include settings for permissions, access levels, restrictions write masks and a text-based description. Table 2 lists attributes of the *Base NodeClass*. Attributes of *NodeClasses* are defined by the OPC UA specification (OPC Foundation, 2017b). Depending on the *NodeClass* of a *Node*, more attributes and reference options are added.

Table 2. Base NodeClass attributes and their DataTypes. Mandatory (M) and Optional (O). (OPC Foundation, 2017b)

Attributes	*Use*	*DataType*
NodeId	M	NodeId
NodeClass	M	NodeClass
BrowseName	M	QualifiedName
DisplayName	M	LocalizedText
Description	O	LocalizedText
WriteMask	O	AttributeWriteMask
UserWriteMask	O	AttributeWriteMask
RolePermissions	O	RolePermissionType[]
UserRolePermissions	O	RolePermissionType[]
AccessRestrictions	O	AccessRestrictionsType

Variables represent values of *Nodes*. In the specification they are separated into *Properties* and *DataVariables*. *Properties* are characterisations of what the *Node DataVariable* represents. Properties are thus used to give meta data to values. They differ from *Attributes* in that *Attributes* give meta data to the *Node*. An example for a property could be an engineering unit of the *Node* value.

DataVariables contain the actual content of the *Object*. This could be any byte data representation such as a number or a file. An example of this could be a temperature as an integer number. This value could then be further described with *Properties* such as engineering unit as kelvins.

References give context to the *Nodes* by describing their relationships to other *Nodes*. These *References* can then be browsed via browsing *Services* explained in Services section of this chapter. References are defined as *ReferenceType Nodes* which are then defined by the *ReferenceType NodeClass*. As seen in Figure 5, Reference form one *Node* to another consists of a *SourceNode*, the *ReferenceType Node* and a *TargetNode*. The target *Node* doesn't need to exist for the Reference to be made. For example, it can point to a *Node* that can be found in another OPC UA *Server* and can be identified by using the *Server* name and *NodeId* given to the *Node* by that *Server*.

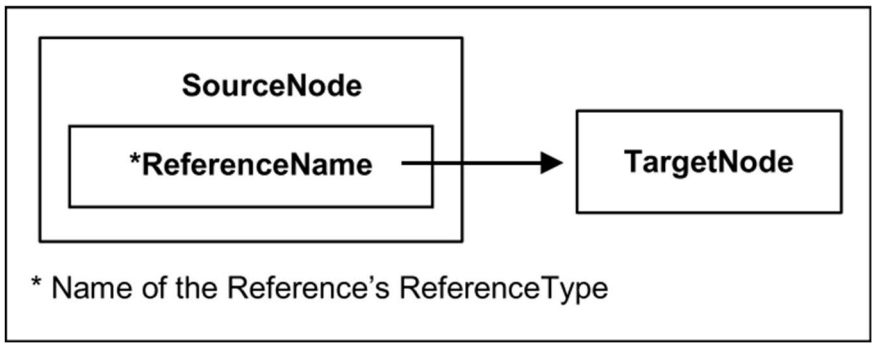

Figure 5. Reference model (OPC Foundation, 2017b).

The specification (OPC Foundation, 2017b) defines numerous different *ReferenceTypes*. At top level the *ReferenceTypes* can be divided into *Hierarchical References* and *NonHierarchical References*. Out of these some are "concrete" *References* that are subtypes of the abstract *References* or other concrete *References*. In Table 3 some of the *References* are listed that are relevant to this thesis's scope. Only concrete *References* are listed as abstract types are mainly used to categorize the concrete references in this thesis' scope. The table as such gives an idea on some of the most common *References* that can be found in an OPC UA Server. In addition to the table, many more *ReferenceTypes* are described in the OPC UA Part 3: Address Space Model Specification (OPC Foundation, 2017b).

Table 3. Common concrete ReferenceTypes (OPC Foundation, 2017b).

ReferenceType	***Hierarchical***	***Description***
HasProperty	Yes	For identifying *Properties* of a *Node*. *TargetNode* is a *Variable* which is then defined as *Property*.
HasComponent	Yes	For identifying *DataVariables*, *Objects* and *Methods* of a *Node*. TargetNode is a *Variable*, *Object* or *Method*.
Organizes	Yes	Organizes *Nodes* in the *AddressSpace*. *SourceNode* can be of either *Object* or *View NodeClass*. *TargetNode* can be of any *NodeClass*.
HasTypeDefinition	No	Binds *Objects* or *Variables* to *ObjectTypes* and *VariableTypes* respectively.

The namespace defines a set of *Nodes* that can be identified with unique *Identifiers* within the namespace. The *Server* can have *Nodes* with the same *Identifier* if they are in their separate namespaces. In other words, the namespace enables having same *Node Objects* with same *Identifiers* across namespaces. *NodeIds* of these objects would still be unique due to having different namespaces. More specifically, the namespace is a Uniform Resource Identifier (URI) that can be found in the OPC UA *Server NamespaceArray*.

An OPC UA *Server* can have multiple namespace URIs which are stored in the *Server NamespaceArray*. *Clients* should retrieve the *NamespaceArray* from the *Server* at start of sessions to ensure that they have the correct *NamespaceIndex* for accessing *Nodes*. The *NamespaceIndex* is the index of the namespace in the *NamespaceArray* and it must be updated on start of session because the OPC UA *Server* can reorder the *NamespaceArray* on boot or if no sessions are active (Unified Automation GmbH, 2019). To access a *Node* in the OPC UA *Server*, the namespace and the *Node's Identifier* in the namespace must be known. With these the *Client* can construct the target *Node's* unique *NodeId*.

Any nodes inside the *Server AddressSpace* can be accessed with a unique *NodeId*. The *NodeId DataType* consist of a *NamespaceIndex*, an *IdentifierType*, and an *Identifier* (OPC Foundation, 2017b). The *IdentifierType* describes the *Identifier* value, which can have one of following types: numeric, string, Globally Unique Identifier (GUID) or opaque. Numeric and string *Identifiers* are self-explanatory. GUID *Identifier* is unique across systems so that

a Node can be tracked moving between OPC UA *Servers*. unique *Identifier* across systems. Opaque *Identifier* is a free-format byte string. Figure 6 presents examples of *NodeIds* with different *IdentifierTypes* (Unified Automation GmbH, 2019). As stated earlier, both *NamespaceIndex* and *Identifier* are required to access a *Node* because two different namespaces could have *Nodes* with the same *Identifier*.

NodeId	
NamespaceIndex	2
IdentifierType	numeric
Identifier	5001

NodeId	
NamespaceIndex	2
IdentifierType	opaque
Identifier	M/RbKBsRVkePCePcx24oRA==

NodeId	
NamespaceIndex	2
IdentifierType	string
Identifier	MyTemperature

NodeId	
NamespaceIndex	2
IdentifierType	GUID
Identifier	09087e75-8e5e-499b-954f-f2a9603db28a

Figure 6. Examples for different types of NodeIds (Unified Automation GmbH, 2019).

2.1.3 Services

This section will detail services that were relevant to this thesis' scope. The OPC UA supports services beyond the ones listed here. The services introduced here consist of Create session, Browse, Read and Write, Add and Delete Nodes. Many more services are supported by the OPC UA server as defined in the OPC UA Part 4: Services -specification (OPC Foundation, 2017c). The services introduced here cover basic functionalities of an OPC UA server that are required for the GraphQL API to provide desired information.

To create a Session between the *Client* and the *Server* a *SecureChannel* must be formed first. This can be done with the OpenSecureChannel service. The secure channel is used with an *authenticationToken* to ensure that the same user is using the Services throughout the Session. The Session can be created once a *SecureChannel* has been made with a CreateSession service. During the creation of a Session, an *authenticationToken* is created and is used in subsequent service requests to the *Server*. Sessions are unique to each *Client* for security reasons and only one session can be made between the *Client* application and the *Server*.

Browse service is used to browse through *Nodes* in the *AddressSpace* via *References* between the *Nodes*. The *References* and *Nodes* that are browsed with this service can be specified in the service request. The *Browse* service request to a specific *Node* returns the *Nodes* that fulfilled the request parameter requirements.

Read service is used to read *Attributes* of *Nodes*. Multiple *Attributes* can be read from multiple *Nodes* with a single *Read* service request. Reading a *Variable* of a *Node* returns a *Variable Object* from which meta data such as timestamps, *DataType* and *StatusCode* can be found without further Read requests. The *StatusCode* describes the usability of the value received from the *Variable Read* with three values: Good, Uncertain and Bad (Cavalieri, Salafia and Scroppo, 2019). The *StatusCode* also includes a SubCode which explains the

quality rating of the value. *Write* service is used to write *Attributes* of *Nodes*. Similar to *Read* service, multiple *Nodes* and their *Attributes* can be written with a single service request.

AddNode service adds new *Nodes* as children to another *Nodes* with *HierarchicalReferences*. The added *Node* will be a *TargetNode* for the *HierarchicalReference*. This means that each *Node* is somehow hierarchically connected to other *Nodes* and the *AddressSpace* is then fully connected. The *NodeClass* of the added *Node* can be specified from existing *NodeClasses* on the server. Additionally, some *Attributes* can also be set with parameters to the added *Node*. *DeleteNodes* service removes *Nodes* from the *AddressSpace*. It may leave *References* unresolved if they had the deleted Node as *TargetNode*. These *References* can be deleted as well if a parameter is set when the *DeleteNodes* service request is made. Deleting a Node that has a child *Node* can leave that child *Node* floating in the *AddressSpace* without any *HierarchicalReference* to other *Nodes* and thus it cannot be found via the *Browse* service. This should be avoided by deleting the child *Nodes* before deleting the parent *Node* or giving new *HierarchicalReferences* to the child *Nodes*.

2.1.4 Ilmatar OPC UA server

Aalto Industrial Internet Campus has a crane called Ilmatar in one of the main halls. Ilmatar crane has an OPC UA server to serve collected sensor data from the crane and to remotely control the crane. The collected data includes current values such as position, load and operations times. The server is accessible from its own private network and can be found as a standard OPC UA server endpoint. Access to the OPC UA server data requires no login or credentials but controlling the crane via the OPC UA server requires a personal access code provided by Konecranes. The access code is written to a specific *Node's* value attribute during control.

There are some less than ideal decisions made on the Ilmatar OPC UA server and one of them is that the *Node identifiers* for sensor data *Nodes* are named in an inefficient way. The *Node identifiers* are used to describe the *Node's* whole folder hierarchy e.g. "Folder1.Folder2.Folder3". Documentation for High Performance OPC UA Server SDK states that using long strings as *Node* identifiers uses up memory and causes slow performance when looking up individual items with *NodeId* (Unified Automation GmbH, 2019). Better solution would be to just use numeric identifiers for *Nodes*. In addition to this, the only source of context for a *Node* is the *Node* identifier which for example could be "Status.Hoist.Position.Position_mm" on the Ilmatar OPC UA server. No *Node* description or engineering unit is used to give context to the value of the *Node* even though they are readily available as part of these Nodes *NodeClass* attributes as specified in the OPC UA specifications (OPC Foundation, 2017b).

Some desired features are also missing on the existing OPC UA server. For example, no Nodes can be created to the server at this moment, which would be useful when retrofitting additional sensors to the Ilmatar crane. Having these retrofitted sensors' data accessible from the OPC UA server would be useful because it would aggregate resources under the same OPC UA communication protocol. Otherwise, the retrofitted sensors' data needs to be retrieved from multiple servers with varying communication protocols. Potentially, sensors could add themselves to any available OPC UA server automatically, which then are accessible from any clients that connect to the OPC UA server. Ala-Laurinaho (2019) has in his work built a sensor configuration platform that makes it simple to retrofit sensors to an already existing system (Ala-Laurinaho, 2019). With the configurator it is possible to remotely define what kind of data the sensor collects and to where it is collected. In Ilmatar's

case it could potentially be possible to pass data to the Ilmatar's OPC UA server, thus adding more sensor data available via it. However, the Ilmatar's OPC UA server does not support creating new nodes by clients for now.

To circumvent this lack of support for retrofitting sensors can be done by connecting another OPC UA server to the same network which would support the missing features. This could be done fairly easily as some libraries, such as FreeOpcUa, readily support starting customisable OPC UA servers (FreeOpcUa, 2018). The FreeOpcUa is also constantly being developed by the community and is currently the most extensive Python and C++ library for OPC UA. Having custom servers also enables the possibility of adding more performance to the hardware when necessary.

2.2 GraphQL

First in this section, the GraphQL query language is introduced. Next, the architecture and usage of a GraphQL server are presented. Then, the benefits and compatibility of a GraphQL API with OPC UA servers are discussed. Finally, the HTTP protocol used with GraphQL is examined, and the GraphQL is compared to RESTful APIs.

2.2.1 Introduction

GraphQL is a query language for clients which also functions as a runtime for fulfilling queries on the application server side. It was first developed and used internally by Facebook. After three years of maturing the language, Facebook released the GraphQL specifications in 2016. As of 2017, the specifications were made available by Facebook under the Open Web Foundation Final Specification Agreement (OWFa 1.0, 2011). Over time, significant amount of web service providers and users have adopted GraphQL (GraphQL Foundation, 2018). Some of these web service providers are big companies, such as Pinterest and Shopify, but Git Hub making their API available through GraphQL has likely made it widely known among developers and thus, helped it spread even further (Torikian *et al.*, 2016).

GraphQL specifications define the requirements for building a GraphQL compliant application server. GraphQL specification does not dictate a language specific code. Instead, application servers can be mapped to follow the specification instructions regardless of what programming language or platform is used. For clients, the specifications describes a query language for making requests and how to handle the resulting data. (GraphQL Foundation, 2019)

GraphQL has some design principles that explain the need and purpose for developing the query language (GraphQL Foundation, 2019):

- Hierarchically structured
    - Congruence with the structure of modern applications.
    - Queries are shaped the same as returned data so that results are predictable.
- Product-centric development
    - Built around the requirements and needs of front-end engineers.
- Strongly typed
    - Each application has its specific type system.
    - Tools are used to ensure syntactically correct queries for the specific type system at any point of a client application development.
- Clients specify the queries
    - Type system is presented to clients who use it to make their own specific queries for consuming the server information.
    - Queries are specified down to a single field eliminating overhead in query responses.
- Introspective
    - GraphQL servers are queryable by the query language itself.

2.2.2 Architecture

GraphQL type structure starts from three root types common to all GraphQL servers. These are also the basic operations that are used to make any queries to the server. *Query* type is used solely to read data from the server. *Mutation* type is used to write data to the server and fetch data with the same query. Finally, *subscription* type is used for receiving data in response to certain events. From these operation types the queries will be branched off via *fields* to more specific queries using more *subtypes*.

The type system is used in GraphQL to describe the capabilities of the server to clients and to validate queries. At lowest level the type is either a *Scalar* or an *Enum*. *Scalar* represents a value such as an integer or string, and *Enum* specifies some valid responses for the field. Two abstract types are also supported by GraphQL. *Interface* defines a list of fields that points to *Scalars* and *Enums*, or more *Interface* and *Union* types. *Union* types defines a list of possible types that can be returned for the field. The collective type system capabilities of a GraphQL server is called a *schema*. Figure 8 presents an example of a GraphQL server *schema* and how data from the *schema* is consumed via a query. With the help of the *schema*, tools like Graph*i*QL can be used to help build and test queries and to present the consumable data from API to users in an understandable format. (Facebook Inc., 2016; GraphQL Foundation, 2019)

Each field in the *interface* of a type has its own resolve function. The resolve function is a bit of code that provides a value for its corresponding field. For example, fetching the value from a database or calculating it based on values in other fields. If the value could not be delivered for some reason, the field will return as ‘null’ and an error will be returned in the ‘errors’ list with the response. The resolve function has access to other types in the schema and should also be capable of returning a list of values or types if the corresponding field is set to return a list.

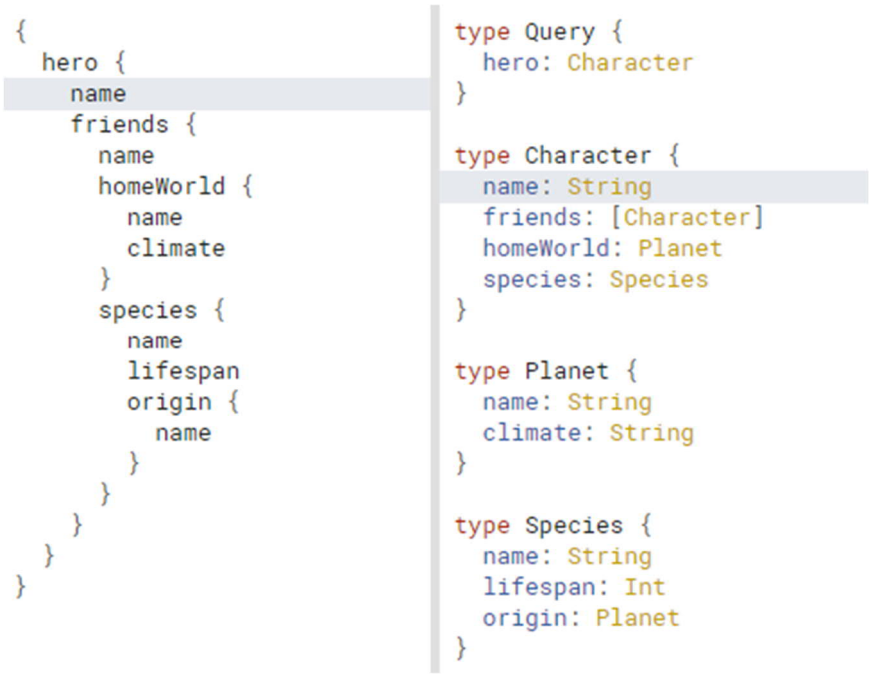

Figure 8. On the right, GraphQL server type system, or schema, from which the fields can be requested by queries such as the one on the left. (Facebook Inc., 2016)

The response from a GraphQL server has 'data' and 'errors' objects at top level. The data includes the actual content of the response and the errors include any field specific errors that occurred during the execution of the query. The data content is similar in structure as the original query, as can be seen in Figure 7. Only the fields from the query are fetched and returned. The structure of the response is predictable as the original query is already structured the same way as the data structure presented by the API.

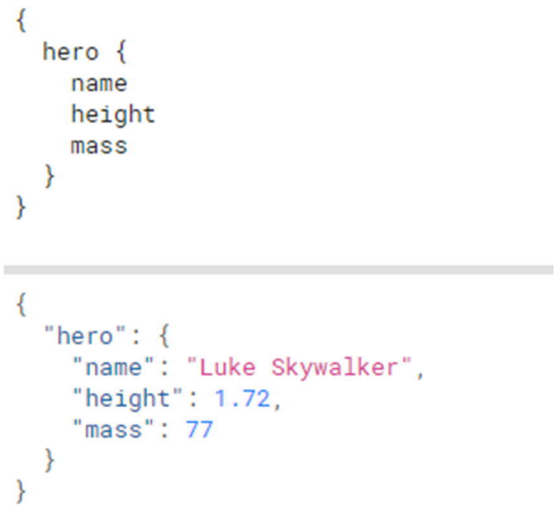

Figure 7. GraphQL query (above) to which a response (below) is returned (Facebook Inc., 2016).

2.2.3 Benefits to an OPC UA server

OPC UA is a powerful tool in industrial communication. However, there is a certain level of knowledge required to properly use and understand the OPC UA Information Model and server communication. The communication with an OPC UA server is enabled by various open source libraries for many different programming languages. Some of the libraries are available on GitHub under the name Free OPC-UA (Rykovanov and Oroulet, 2014). These libraries and other open source projects are often the best resource, in addition to the OPC UA specification, for learning how to programmatically communicate with an OPC UA server.

Learning to communicate with an OPC UA server can be time consuming and difficult. Firstly, learning from the libraries and open source projects requires good knowledge of the programming language used as you may need to thoroughly study the code. Secondly, the OPC UA specification can be overly extensive beyond requirements for general usage, and only gives guidelines on how the communication should be done with no practical examples. Because of these difficulties in learning the communication with OPC UA servers, an easier to learn and use API would be beneficial. The API could present the OPC UA Information Model in a way that any developer who is familiar with its architecture can start developing applications for the OPC UA server data at relative ease. This in extension means more applications developed for industrial systems from non OPC UA experts. GraphQL is a growing API language and its use is identical between any GraphQL server. Only understanding the data model needs some reading, but GraphQL has methods to present it to developers dynamically with the aforementioned type system, schema.

As mentioned earlier, Facebook claims GraphQL is built based on the needs of front-end engineers. This resulted in GraphQL data being presented in a tree like object structure. Coincidentally, OPC UA Information Model is also a tree like object-based structure. This means that variables would be the leaves of the object structure tree just like in GraphQL the Scalars would be the leaves. Potentially, a GraphQL API could be modelled to exactly present the Information Model of an OPC UA server. Different types of OPC UA nodes can be presented as types in GraphQL schema and references between nodes would be the fields of the type interface. With the GraphQL schema the OPC UA Information Model can be represented to GraphQL API consumers with all relations to other nodes and variables staying intact. Lastly, an OPC UA read and write service supports reading or writing values from multiple nodes and attributes with a single request instead of making a unique request for each attribute. This feature can be taken advantage of with a GraphQL dataloader which collects requested fields from an API query and batches them together to send only a single request for values to the OPC UA server (Byron and Contributors, 2015). This should result in considerably faster queries in slow internet conditions. Query execution in GraphQL API section explains the function of this feature in more detail.

2.2.4 HTTP

Hypertext Transfer Protocol (HTTP) is one of the most common application layer protocols. HTTP is mainly used for accessing resources around the internet and has been a fundamental part of the World Wide Web data transfer since 1990 (Fielding *et al.*, 1999). HTTP is based on requesting data from a server and then waiting for a response. No session between the client and the server is managed beyond a single request-response interaction. Thus, no session information is stored on either end between unique requests. Further requests are dealt as a new client connecting to the server. This is due to HTTP being a stateless protocol.

Any kind of data is possible to transfer with the HTTP protocol as long as the recipient is capable of handling the content. A downside to HTTP protocol is that it has slow data transfer speeds and high overhead, and is considered to be one of the slowest when comparing it to other application layer protocols for IoT applications (Ala-Laurinaho, 2019).

HTTP communication is done over a Transmission Control Protocol (TCP) connection which is a transfer layer protocol. TCP is used to provide a reliable and ordered data exchange between two hosts over an IP-network. The TCP connection is initiated with a TCP three-way handshake, or TCP-handshake. TCP-handshake sends three messages called Synchronize, Synchronise-Acknowledge and Acknowledge between the connecting hosts to begin a TCP session. The TCP-handshake is used to negotiate the connection parameters. Once the connection has been established, data transfer is based on sending segmented data packets and receiving acknowledgements for the packets. TCP orders the sent segmented packages with sequence numbers. If a packet is lost somewhere along the transmit or no acknowledgement for the packet is received, the packet can be resent by the sender. After the data has been transferred, the connection between hosts is terminated. For following data transfers a new connection is established. (Kurose and Ross, 2013)

GraphQL API queries are generally done with HTTP requests. A HTTP request structure consists of a start line, headers, and a body (MDN Web Docs, 2019b). An example of a HTTP request when requesting a GraphQL API can be seen in Table 4. The start line defines what method is used, what resource is targeted on the host, which port is used, and what HTTP protocol version is used. HTTP request can use multiple predefined methods to indicate what operation is requested for the target resource. Some of the common methods are GET, POST, PUT, PATCH. Headers are used to pass additional information with a HTTP request or response (MDN Web Docs, 2019a). General headers pass information of both the request and the response. Request headers contain information on the target resource or of the client requesting it. Entity headers pass information regarding the request body. The request body holds the data passed to the server, in this case, the GraphQL query. Depending on the method used the body might not be included in the request.

Table 4. Example HTTP request structure for a GraphQL API query.

Start line		POST /graphql HTTP/1.1
Headers	***Request headers***	Host: www.graphql.api User-Agent: Mozilla/5.0 (Windows…
	General headers	Connection: keep-alive Upgrade-Insecure-Requests: 1
	Entity headers	Content-Type: application/json Content-Length: 390
Blank line separator		
Body		[GraphQL query in JSON format]

A HTTP response consists of a status line, headers and a body. An example of a HTTP response can be seen in Table 5. The status line contains information of the protocol version, status code and status text. Status code indicates either a success or failure of the request. Status text is a description of the status code and is meant to give information for understanding the HTTP message. The headers, much like in the request, pass information

of the server, the response and the body. Additionally, cookies can be sent within the response headers to be used for later requests. The response body includes the GraphQL response to the query. (MDN Web Docs, 2019b)

Table 5. Example HTTP response structure for a GraphQL API response.

Status line	HTTP/1.1 200 OK
Headers	Access-Control-Allow-Origin: * Connection: Keep-Alive Content-Type: application/json Date: Fri, 10 Sep, 2019 12:34:56 GMT Server: WSGIServer Set-Cookie: csrftoken=lFdFjr… Vary: Cookie, Origin X-Frame-Options: SAMEORIGIN
Blank line separator	
Body	[GraphQL query results in JSON format]

GraphQL queries are commonly made using solely HTTP POST method targeting /graphql resource. GraphQL does not take the method or target URL in consideration when determining the resource and operation. Instead, the target resource, operation and other information are found in the GraphQL query passed within the HTTP request body.

2.2.5 Comparison to RESTful

GraphQL and REST are the two most popular web API technologies used today. While they have their own design differences which might make them suitable for different tasks, they are both generally suitable for any web API needs. Some design choices, however, can help make the difference in what technology is more suitable for the task at hand. These stem from built in features, available library extensions, and the availability of the documentations for both.

The main similarity between the GraphQL and REST is that both utilize HTTP requests and return a response in some form. The difference between the technologies' HTTP requests is that REST makes use of different HTTP methods such as GET, POST, PUT or UPDATE. GraphQL needs only POST method with a JSON query, which defines target resource and operation, in request payload. Many of the REST elements such as the HTTP methods are built in to the GraphQL library and correctly executed on the server side. (Stubailo, 2017)

Both GraphQL queries and REST requests end up calling some function on the server side which then returns data accordingly (Stubailo, 2017). The called function in REST is dependent on the request URL and the chosen HTTP method. The function will then return a developer defined response for that specific endpoint. With GraphQL, multiple function calls can be made with a single request that can be found in the HTTP body. The amount of functions called depends on the amount of resources accessed and the specified fields of those resources. Each field in a resource calls its own resolver function which can be customised server side to return a value for the field. Sometimes, a GraphQL API returns a nested response of multiple resources depending on the query. Overall, with GraphQL, the client knows what field values it wants to retrieve, but does not know how each field's value

is in fact resolved on the server side. Finally, the GraphQL response structure is defined by the GraphQL library and is dependent on the queried resources and fields.

With REST, the aforementioned function call, dependent on the URL and HTTP method, always returns a server-side defined response for that specific resource. If multiple different resources are needed, the client needs to make multiple HTTP requests for each resource (Stubailo, 2017). In addition to that, the REST API might return values that are not needed by the client from a resource causing unintended traffic between the client and the server. With GraphQL the client's query structure defines precisely which resolver functions are called, and what resources and fields are returned within the response. This is possible because each field in a GraphQL API has its own independent field resolver function. This can potentially save traffic over the network and also save time when unwanted field values are not resolved on the server side. Retrieving multiple resources with a single query from a GraphQL API can be done by chaining the resource queries. However, query chaining is limited only to either reading or writing to resources. Independent resolver functions also enable hierarchical object fetching, as in retrieving another resource which has some relation to its parent.

When comparing how the data is represented by the API technologies to the client, some important differences can be found. GraphQL API is capable of showing the whole OPC UA Information Model with all the relations and depth intact. In contrast, a simple REST API would present the Information Model as flat with no obviously visible relations with requested node to other nodes or values. Thus, it would be preferable to use a GraphQL API in this case as it doesn't hide the potentially important context for the data consumed by clients.

2.3 GraphQL API

This API for Ilmatar OPC UA is developed to increase its usability. Directly communicating with the OPC UA server requires in depth understanding of the OPC UA architecture. In addition, the communication is somewhat dependant on the programming language and the platform used. A solution to this would be using a common web-based API technology to ensure that the OPC UA is accessible from any platform. A web-based API would be accessible anywhere within the same network. To build this web-based API, REST and GraphQL API's were considered. These technologies are already widely used in popular web platforms. Out of these options the GraphQL was selected as it provides a platform that is simple to build custom solutions on and views the resource a bit similarly to the OPC UA AddressSpace architecture. Both API technologies can fulfil the requirements set for the web-API, but GraphQL has more built in benefits for this use case, as was discussed in the previous section. With libraries, GraphQL and REST are supported by many different programming languages and web frameworks.

The built GraphQL API provides a new access point for retrieving and writing data to the Ilmatar OPC UA server. This gives developers an option to use either the simpler GraphQL API or the more complicated OPC UA server directly. The GraphQL API works as a separate server in the same network as the OPC UA server. The API communicates as a client with the OPC UA server. Thus, the API will have access to the same OPC UA views and nodes as any other clients have access to. This helps in development with the fact that the API doesn't need any extra security on top of the existing security features on the OPC UA server. The API does not need to know anything of the inner workings of the OPC UA server and,

thus, uses standard service requests to read and write data, just like a normal client would do. The API can aggregate multiple different OPC UA servers under the same API. Clients using the API provide identification information for the target server and node within query arguments.

Researchers have developed API's directly to OPC UA servers before (Grüner, Pfrommer and Palm, 2016). These solutions should perform better in terms of latency and efficiency compared to a separate API server due to not having to act as a client with the OPC UA server. Instead, the API would be able to access files directly server side without additional connection management. However, these server-side APIs are harder to configure and won't work with OPC UA servers to which the developer has no complete access to. A separate server can be configured with only client end point address to the OPC UA server and should work with most existing OPC UA servers as it is considered by the server as a client. In the case of Ilmatar it is preferable that the API would be separate from the OPC UA server as it would ensure that any expansions to Ilmatar OPC UA servers would also be supported by the API with little configuration. Cavalieri *et al.* (2019) have developed a RESTful API server that functions as a separate web platform between the user and the OPC UA server (Cavalieri, Salafia and Scroppo, 2019). They built the web platform to provide a friendly way to communicate with the OPC UA server and to simplify the view of the OPC UA Information Model. This and other existing web APIs for OPC UA are mostly RESTful APIs. This is partly the reason a GraphQL API is developed in this thesis project instead of a RESTful API as it may provide some more useful features that are not present in the RESTful APIs.

2.3.1 Requirements

The first and most important requirement for the API was that anyone could start developing for it with little effort. Thus, a commonly used API architecture was required. GraphQL is a new but emerging alternative for REST. It provides some features that are hard to create on a RESTful API. Many big web development companies, such as Facebook and GitHub, have added GraphQL support to their web site APIs (Torikian *et al.*, 2016). GraphQL has originally branched off Facebook and they have made an extensive documentation for the query language (GraphQL Foundation, 2019). They also likely have the capability to properly develop the API usability to the level required for common developer adoption. This leads to a GraphQL API fulfilling the first requirement.

To ensure responsive communication with the GraphQL API, the number of requests should be minimized between the client and the API and between the API and the OPC UA server. This means retrieving as much information as possible with single request, but the API should not over fetch data from the OPC UA server either. Query chaining, provided natively by GraphQL, enables access to multiple different resources with one single request. The desired resources are gathered to a list that makes up the query in the request body. In addition to query chaining between the client and the API, also the communication with the OPC UA server should be minimized to reduce latencies. OPC UA server communication should be minimized either by accessing only desired resources, batching requests between the API and the OPC UA server, or using caching.

The API should be scalable to wrap multiple OPC UA servers. The OPC UA server on Ilmatar does not support creating nodes for retrofitting sensors. This can be circumvented by adding another OPC UA server parallel to the Ilmatar OPC UA server. Sensor data from both OPC UA servers should then be accessible via the same GraphQL API. Preferably the

resources on both OPC UA servers should be accessible seamlessly, as in the client would not need to know on which OPC UA server the sensor data can be found on.

The API should have support for reading, writing and creating nodes and their values. Clients should have access to data defined in Table 6. Read access should be available for any relevant data that could be used outside of the OPC UA environment. Write access should be at least for any input fields that can be used for operation of the Ilmatar crane. Creating nodes should be possible for creating folder like node structure and adding variable nodes for sensors to input their real time values to.

Table 6. OPC UA Node operations to be supported by the API.

Reading	***Writing***	***Adding Nodes***	***Subscription (optional)***

• DisplayName
• Description
• Variable
  o DataValue
  o DataType
  o SourceTimestamp
  o StatusCode
• NodeId
• Child Nodes
	• DataValue
	• Description
		• Folder Node
		• Variable Node
			• DataValue

Finally, a nice to have feature for the API is to have an option to subscribe to certain *Node* value changes. This means that clients can get notifications when a monitored *Node's* value is changed past a threshold. For this to work, the API would need to be able to itself subscribe to the *Node* value on OPC UA server, and on notification forward it to the client who set up the subscription.

Specific API requirements in short:

- Commonly used and extensive documentation.
- Minimized data traffic between client and API, and between API and OPC UA server.
  - Accessing only requested resources.
  - Query chaining between the client and API.
  - Batch requested data to OPC UA service requests.
- Scalable to cover multiple OPC UA servers.
  - OPC UA servers configurable via the API.
- OPC UA services to support.
  - Reading relevant *Node* attributes.
  - Writing to values *Node* value and description attributes.
  - Creating and deleting folder and variable *Nodes.*
  - Subscribing to *Node* value attribute changes (optional).

2.3.2 Architecture

The GraphQL API server acts as an independent entity in the Ilmatar crane network. Clients can communicate with the API server without authentication just like they can do with the OPC UA server. Although the OPC UA server requires a session for communication to take place, it is handled between the server and the API. Thus, no session management is required between the client and the API. An example of the communication between servers can be seen in Figure 9. In the figure the red line illustrates the first query flow when retrieving data from the OPC UA server. Queries are sent to the GraphQL API which transforms them into service requests that are understood by the OPC UA server. The API establishes the secure channel and activates a session for the OPC UA server only during the first query to the API. The following queries use the existing session and thus will be executed faster.

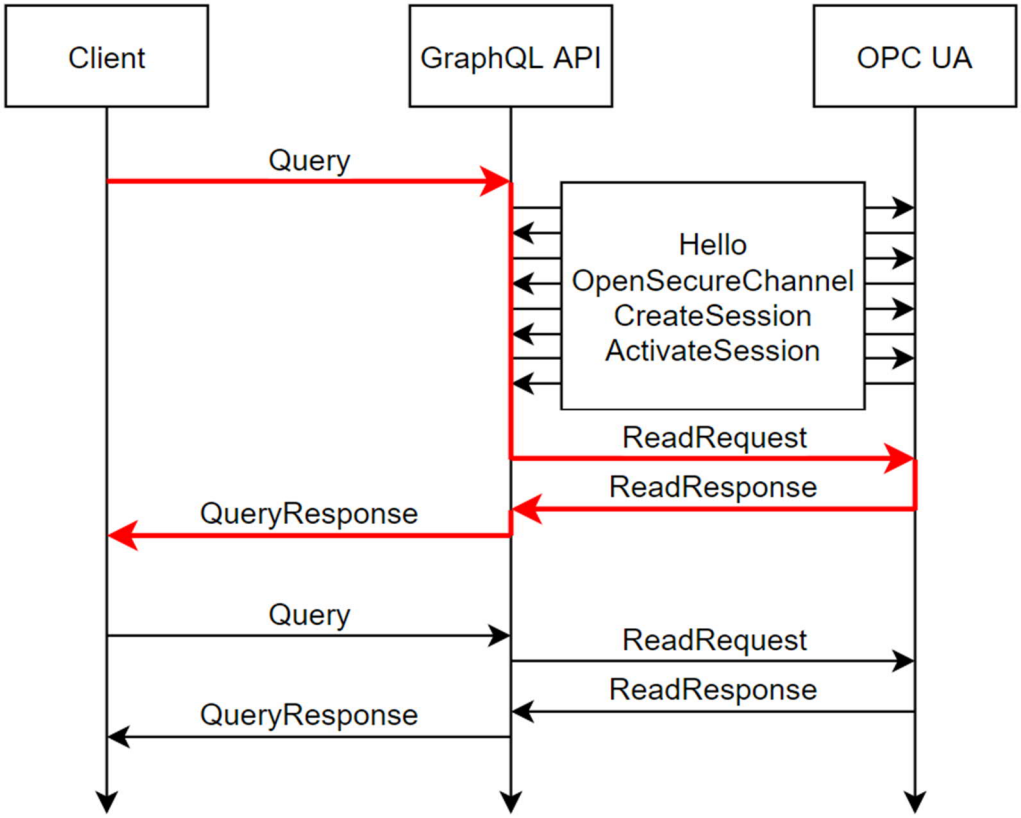

Figure 9. Communication flow between the Client, GraphQL API and OPC UA server.

The amount of communication between the API and the OPC UA server depends on how complex the queries to the API are. As explained in the OPC UA Services part, OPC UA server can receive multiple nodes and attributes in a single read or write service request. However, retrieving targets of references such as 'subnodes' of a node cannot be combined with these read or write service requests. Retrieving references requires its own type of service request from the OPC UA server. Thus, if some node's subnodes are requested in addition to an attribute, the GraphQL API will have to make two different service requests to the OPC UA server. Simple queries such as requesting multiple node's attributes can be completed with only one service request to the OPC UA server.

2.3.3 Queries and responses

The requests are sent to the API using HTTP POST method with GraphQL queries included to the payload. The query payload is transformed into JSON format before sending. The API returns a response with the queried data also in JSON format. Queries are constructed following the standard GraphQL language. This API supports queries and mutations. The queries are used for reading OPC UA node attributes and hierarchical references. The mutations are used for setting node value or description attribute, and for adding and deleting nodes. Added node type is dependent on the arguments given by the client. Clients can add either a folder type node, which organizes nodes, or a variable node which has a value representing, for example, a sensor value.

In practice, queries are written in format as presented by Figure 10 which is an example of a simple node value query. First, the client defines what type of query method the user uses, in this case 'query' which is used for reading data. Next, client chooses one of the possible subtypes (*node*) for this query method and writes required arguments for it (*server* and *nodeId*) that specify which node attributes are read from. The client can then choose which fields (attributes) are requested from the target node (*name* and *variable*). Some fields such as the variable field have more subfields that can be specified in the same manner as before (*value* and *datatype*). Due to independent field resolving, only fields that are found in the query are fetched from the OPC UA server. Thus, removing unnecessary fields form queries can reduce the amount of time taken to receive the response.

```
query {
  node(server: "Ilmatar", nodeId: "ns=7;s=SCF.PLC.DX_Custom_V.Status.Hoist.Position.Position_mm") {
    name
    variable {
      value
      dataType
    }
  }
}
```

Figure 10. Example query for the GraphQL API.

The query arguments for node type are specific to the instance of GraphQL API and OPC UA server queried. The *server* argument is the name of the OPC UA server as specified in the GraphQL API settings. It has nothing to do with the actual OPC UA server, but is only used to separate the servers in the API layer. The server names are set in the GraphQL API settings when also adding the end point address for the OPC UA servers. The *nodeId* argument is the same that the OPC UA server uses to address specific nodes. In this OPC UA server's case it consists of the *namespaceIndex* and string identifier. Table 7 presents some examples of *nodeId's* possibly used in queries and explains the formation of the *nodeId* for the *nodeId* argument. As it can be seen in the Table 7, the NodeId string variates based on the *namespace* and the type of *identifiers* used on the OPC UA server. Nodes and their *NodeIds* can be browsed on the index page of the GraphQL API. Browsing to the desired node reveals the *server* and *nodeId* arguments needed for accessing attributes in of that browsed node. In mutation queries, more arguments in addition to node identification are required. These could include for example the value argument which is to be set to the node or a parent *nodeId*, which is used when adding a new node to the server. More info on the query arguments, fields and types can be found in the GraphQL API Graph*i*QL documentation.

Table 7. NodeId components and their stringified versions used as query arguments.

NodeId components			*NodeId string*
Namespace Index	***Identifier Type***	***Identifier***	
7	string	Status.Hoist.Position	ns=7;s=Status.Hoist.Position
7	numeric	1234	ns=7;i=1234
0	numeric	1234	i=1234

The responses are returned in similar format as the original query, and they are also in JSON format. Figure 11 has an example response to the query found in Figure 10. The response can include both *errors* and *data* keys. *Errors* key is only present if errors were raised during the execution of the query. *Data* key has all the fields and their values in the same exact format as in the original query's *node* field.

Figure 11. Example response to the query seen in Figure 10.

2.3.4 Query execution

Once the GraphQL API receives a query from a client, it will generally follow the flow example presented in Figure 12. Based on the query root type and subtype, the query is first routed to its corresponding resolver function. The resolver function receives the arguments that are given for the node subtype as seen in Figure 10 (i.e. *server* and *nodeId*). The function uses the node *server* argument to find the correct server object from pre-constructed server objects. Each server object manages its own connection to its target OPC UA server. This in combination with each query using the same server objects ensures that only one connection is established between the GraphQL API and each OPC UA server. If the resolver function finds the correct server from the server list, it initiates the *node* subtype object to which is given the target server's name and the node *nodeId* argument as initial values.

Once the *node* subtype object is initiated, each requested field from the query runs its own resolver function. If the field value needs to be retrieved from the OPC UA server, the resolver function will make a call to a dataloader. The dataloader waits for calls from the resolver functions and then batches them into a single service request for the OPC UA server. Each OPC UA server that is accessed is required to have its own service request. In the case of requesting subnodes of a node (i.e. *browse service request*), a dataloader is not used. In terms of performance it is recommended to retrieve data from each node with its own *nodeId* and subtype. However, even if the dataloader is not used for retrieving subnodes, it is

possible to be implemented in the future. OPC UA specification supports retrieving references from multiple nodes with a single service request (OPC Foundation, 2017c).

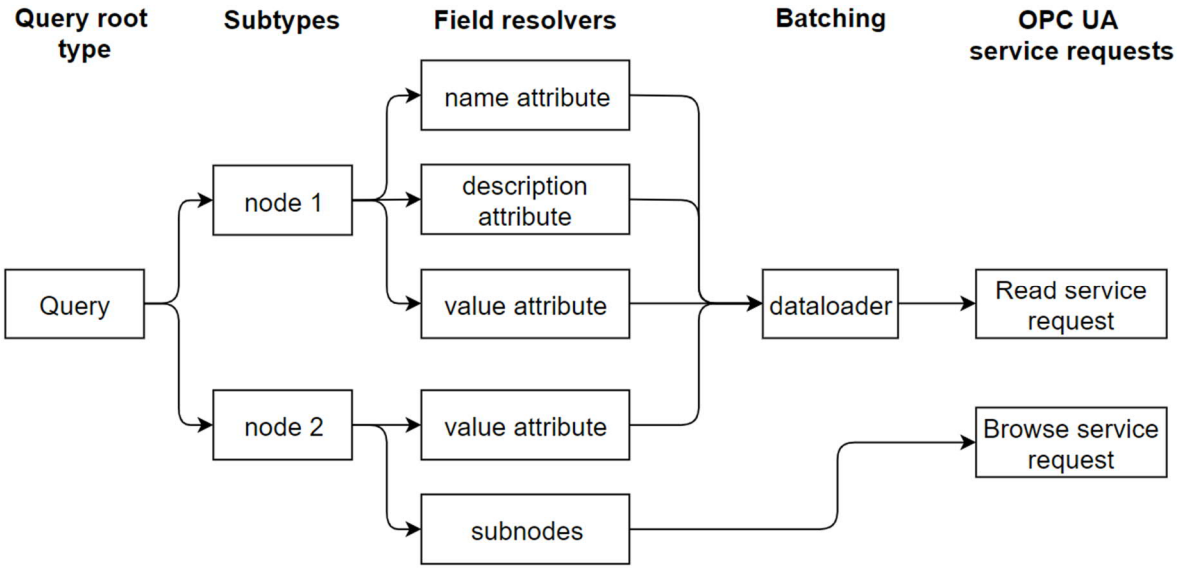

Figure 12. Query execution example in the GraphQL API backend.

After the dataloader receives response from the OPC UA server, it returns values back to their corresponding field resolver functions. Field resolvers now have data to be returned for their fields. The data is now structured for returning in response payload. Finally, the response is sent back to the client that made the original query.

2.3.5 Software

The GraphQL API was built with Python programming language as it is quick to code on and its syntax is considered elegant and easily understood from just reading the code (Sanner, 1999). Python is available on practically any platform which makes it a good choice for applications that could potentially be used on multiple different platforms. It also has a large community that supports it with open source libraries. The open source library for Python used for communicating with the OPC UA server is Free OPC-UA library (Rykovanov and Oroulet, 2014). The library is freely available on GitHub and simplifies a lot of communication between a Python client and an OPC UA compliant server. However, some library functions had to be manually modified for optimization and requirement purposes.

The GraphQL application server is built on to Django Framework which is a popular Python based framework for creating web applications (Django Software Foundation, 2018). Django Framework provides a backend with URL routing, database organization and security features so that developing a web application on top of the framework is effortless and secure. GraphQL can also be built on top of Django Framework with existing GraphQL Python libraries of which some are directly made for Django (Haro *et al.*, 2016). Graphene Django library provides tools to integrate GraphQL to your existing Django server with full support for Django database models. This project will not use the Django database integrations from the library as it uses existing OPC UA servers as databases which require the use of the Free OPC-UA library.

The GraphQL server is run on a Raspbian operating system but can be run on any platform once required software dependencies are installed. Raspbian is based on Linux Debian operating system and is specifically developed for Raspberry Pi (Raspbian.org, 2016). The

Django application is run by a Gunicorn server. Gunicorn is a Python Web Server Gateway Interface (WSGI) HTTP Server for UNIX based operating systems (Chesneau *et al.*, 2016). The main function of the Gunicorn server in this system is that it provides and manages multiple workers that execute queries from clients. The number of workers that can be used depends on the cores and threads that the Raspberry Pi CPU has. The Gunicorn receives its queries forwarded by a proxy server NGINX. NGINX server listens to HTTP requests for the Raspberry Pi from the network and forwards them to correct applications (NGINX, 2017).

The GraphQL type interface fields can be customised to retrieve data from practically any combination of data sources. Each field is resolved independently and thus one field could retrieve data from one data source while the next field under the same type could retrieve data from completely different source. This feature used together with the Free OPC-UA library enables the GraphQL API to serve data from multiple different OPC UA servers instead of having to use built in Django databases. Usually though, the data will be served from a single OPC UA server. In this case a GraphQL compatible dataloader can be used to help batch together multiple OPC UA Read service requests (Byron and Contributors, 2015). The dataloader in this project collects attribute read requests from field resolver functions and batches them together into a single read request for the OPC UA server. This results in reduced traffic between the API and the OPC UA server which in turn should reduce overall API usage latencies. The complexity of the query affects how much of the attribute read requests can be batched together in the dataloader.

2.3.6 Hardware

The GraphQL API as a proof of concept is run on a Raspberry Pi 3 model B+. The relevant features of the Raspberry Pi are 1.4GHz quad-core processor, ethernet cable port and possibility for a UNIX based operating system: Raspbian (Raspberry Pi Foundation, 2019). The device is also easy to install to the crane's electrical boxes as it does not take much room. Better performance for the API could be achieved with faster devices, but as a proof of concept the Raspberry Pi is considered good enough. The Raspberry Pi can be connected to the Ilmatar crane network via an ethernet cable or Wi-Fi. Preferably with an ethernet cable as Wi-Fi can be unreliable. The clients that are connected to the Ilmatar crane network should be able to connect to the GraphQL API server via its IP-address.

Figure 13 has a map of connections between the devices which are related to this thesis work. Client is connected to the network via Wi-Fi. From the Ilmatar network, GraphQL API, OPC UA server and Control application are accessible. GraphQL API and OPC UA server are located onboard the crane in electrical boxes. Inside the electrical boxes the servers are connected together via a Wi-Fi router with ethernet connections. This is to ensure good quality connection between the GraphQL API and the OPC UA server. The external network hub is located in the same hall as the crane and is connected to the crane via a Wi-Fi connection. It is easy to plug in additional servers to the Wi-Fi extender in the external network hub which would then be accessible from same network as the servers onboard the crane. The control application which is used as a case study in this project is also connected to the external network hub via an ethernet cable. The client can download the control application from the Control application server and use it to give commands directly to the GraphQL API with the same Wi-Fi connection.

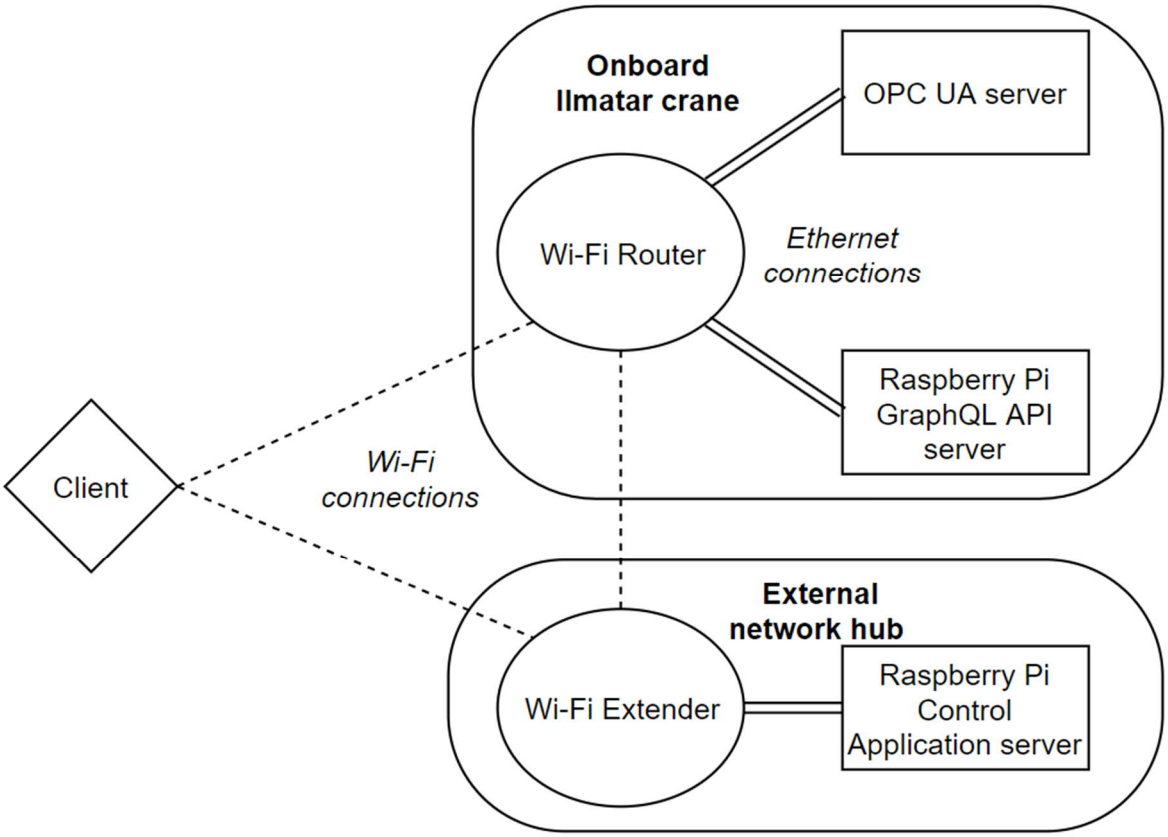

Figure 13. Connections between the devices in the Ilmatar crane hall.

2.4 Case study: Web-based control application

As a case study to test the web interface, a web-based control application is made. The application both reads and writes to the OPC UA server via the GraphQL API multiple times per second. These read and write requests include incrementing a "watchdog" value, reading values related to the "watchdog" status, reading sensor values, and sending control signals to the crane. The application is used in tests to determine whether the GraphQL API provides enough performance for the control application to feel responsive to user inputs. Especially, when data is exchanged continuously in the background on top of the user inputs.

The control application is a web application that manages control access to Ilmatar crane and serves different control schemes. Controlling the crane is done by user requesting the controls from the web application. Multiple users are allowed access the website at the same time, but only one user at a time will be allowed to control the crane. By default, the application serves basic directional controls. However, additional control logic scripts can be added for the web application to serve to users. The control logic can use sensor values from any web resources accessible from the script. The Ilmatar crane's sensor values can also be monitored with the web application. This does not require control access and thus can be done by any number of users simultaneously.

The application is developed with mobile platforms in mind. The layout is scaled with the screen aspect ratios, and touch controls are used to control the crane and navigate the application. Figure 14 has an example of the crane control user interface. Different monitoring logics can be accessed from the tabs at the top. On the selected monitoring tab, user can add monitored sensors from a drop-down list. The monitored sensor values can be

Figure 14. Web-based control application user interface.

either updated manually or set to update automatically. Below the monitored sensors are options to both request (hidden in figure) and release control of the crane. Once a control request is approved by the application server backend, directional controls are shown to the user. Small indicator next to the buttons indicates both when the user has the crane control permit via the application and when the crane is ready to receive control instructions from the application. Directional control buttons are hidden if the user has not requested access for control or someone else is currently controlling the crane. Symbols on the directional buttons match the symbols printed on the crane that instruct which direction the crane moves with each button press. Symbol buttons move the crane parallel to the ground while up and down arrows move the crane hook up and down.

The application is fully web-based. As such, JavaScript seems to be the obvious choice for the programming language of the application's logic. JavaScript is a client-side scripting language widely supported by most web browsers (W3Schools, 2016). JavaScript also has highly developed libraries for fetching data from web resources via HTTP requests. Because of this, retrieving data from the GraphQL API is simple. Additionally, web applications have flexibility in that the application can be run without any installation required on most devices with a browser.

2.4.1 Requirements

The main purpose of the app, in this thesis, is to be a case study for testing the viability of the developed GraphQL API in different applications. These applications have varying needs for data and data update frequency which tests the API's performance. The control application also has a purpose beyond being a case study for the GraphQL API. The application is planned to be a part of the Ilmatar's digital twin. Thus, there are some requirements that specify how the application should function.

The application should be web based and mobile friendly. This means anyone with a browser capable device is able to connect to the application server and use it. Most users have a mobile device already at hand and as such, the application is accessible by virtually anyone. This also gives the application a benefit of being decentralized. Multiple users can access the crane controls without the need of physically handing over the controller.

Multiple people having simultaneously access to the crane controls may raise some safety concerns. Multiple people should not be able to control the crane at the same time. The solution to this does not need to be fool proof, but should prevent accidental overlapping control. Control access is to be requested by users and only one user at a time will have access to the controls. In addition to this, the Ilmatar crane has already several safety concerns thought out. For the crane to move with the commands sent to the OPC UA server, a watchdog variable must be incremented constantly. If the watchdog variable fails to increment, the crane will halt movement and stop listening for commands from the OPC UA server. This ensures that the crane will halt movement in case the control application crashes. In addition to the watchdog, a dead man's switch can be found on the radio controller. The switch must be pressed down when controlling the crane via the OPC UA server. Otherwise the crane will not move. The control access management and these existing safety features should be properly implemented in the application.

When controlling the crane, a personal access code is required. The access code is used to track who's using the crane and some access codes might have some limitations when controlling the crane. The application must prompt user for the access code and forward it to the OPC UA server during control.

In addition to above requirements, some functionalities were requested for the application. These functionalities are intended to load the GraphQL API in different ways. Firstly, a direct control of the crane is required. The direct control tests how responsive the application is to use when controlling the crane via the GraphQL API. Secondly, a customisable set of sensor values must be possible to monitor during the control. These can cause load because the monitoring queries are slightly larger for each monitored sensor value requested. This might cause high latencies especially in weak or slow Wi-Fi connections. Lastly, the application should be able to integrate more custom logics after the application's initial development has finished. For example, a new custom logic should also be able to get data from the crane's position values and also send control signals to the crane based on the logic or user input. This may require the control application to retrieve and send data at potentially high frequencies. It should be possible to be retrofit these custom logics with little effort from the developer. This last feature is requested as the application is planned to expand into a more general Ilmatar crane web user interface, where developed control and monitoring logics are accessible from a single web interface. This web interface would then be accessible from any mobile device and thus is easy to present in demos.

An end goal for the control application is to be a dynamic part of the Ilmatar digital twin. To achieve this, it should be easy to add and remove monitoring and control logics. This way it can function as a platform for integrating JavaScript applications to the digital twin. The application can in the end be potentially seen as a user interface to a part of the Ilmatar's digital twin.

2.4.2 Software

The control application is done with Flask backend (Pallets, 2010). Flask is designed for small scale web applications which use Python as backend programming language. The backends' purpose for this application is to serve the frontend application and manage user access to the crane controls. Thus, the backend is lightweight compared to the GraphQL API backend. Flask was a good fit for this applications purposes as it does not provide unnecessary features in the core package.

The application frontend consists of a HTML web page which is heavily supported by JavaScript scripts. All the logic features on the web application frontend are done with JavaScript. These include fetching specific control and monitoring logics from backend and managing user access to crane controls on the client side. JavaScript enables the web application to be run on most devices that have web browser capabilities. Thus, the user could be running the application on virtually any mobile device or computer. In addition to this, the application is accessible by visiting a server IP-address if user is connected to the application server's network.

3 Results

The results section consists of the GraphQL API's and Web-based control applications benefits to the Ilmatar's digital twin, and performance of the GraphQL API. The performance section measures the performance of the GraphQL API when used with the case study control application. Performance results are compared to directly using the OPC UA server.

3.1 Benefits to Ilmatar's digital twin

The GraphQL API gives some significant benefits to the Ilmatar's digital twin. All the data on Ilmatar's OPC UA server is accessible via the GraphQL API. Thus, developers do not need to know how to communicate with an OPC UA server, but can retrieve all the data via the GraphQL API instead. Another benefit is adding the support for retrofitting sensors to Ilmatar. The Ilmatar's OPC UA server does not support adding new nodes for sensors programmatically, but the GraphQL API can aggregate multiple different OPC UA servers under the same interface. This means that another OPC UA server can be installed to Ilmatar that supports this feature of adding nodes. Both servers can be then be reached via the same GraphQL API.

The case study control application has also brought some useful features for Ilmatar and its digital twin's future development. The application is designed for implementing different control and monitoring logics with access to various sensor data on the Ilmatar's OPC UA server and other resources. These control logics can be easily added to the control application and requires the developer to only know how to use the popular JavaScript programming language. JavaScript on mobile device browsers is great for developing interactive applications especially because mobile phones are commonly fitted with various sensors that can be utilized in the script. This means that developers can make applications that use sensors on the mobile device. With these sensors, for example, an application can be made that tracks the user's phone's position via GPS and makes the crane follow it. Other options could include using the device's accelerometer in an interactive control scheme. Some common mobile device sensors that can be of use are the touch screen, GPS and accelerometer.

3.2 Performance

The GraphQL API's performance is tested using the case study's control application in both controlled and real-life environment. The test is done by sending multiple times a value change request to the API with the control software and then measuring the time that the requests take to receive a response. This method attempts to measure the request execution times in the control software, the GraphQL API, and the OPC UA server. Some differentiation can then be done to specify how much latency is caused by each part of the software communication chain. Tests are done by requesting both the GraphQL API and the OPC UA server directly.

The request that is used for testing changes two values on the OPC UA server. The direct OPC UA requesting test also changes two values, but the request type is different. However, both requests to GraphQL API and OPC UA server directly aim to do the same by changing two values on the OPC UA server. No caching is used to ensure that the requests are completed identically each time.

3.2.1 Controlled environment

Controlled environment tests are done by running the control application, GraphQL API and OPC UA server internally on the same computer. The OPC UA server is run from a python script that is found as an example in the python-opcua library Git repository. The GraphQL server is run on Django development server which is primarily meant for internal testing. All servers are run on single threads, meaning that no parallel execution is done in GraphQL API or OPC UA servers. The control application, which will also function as a test application for the GraphQL API is written in JavaScript. When stated, the test application runs a watchdog incrementor in the background to simulate a real crane control situation. The watchdog is run on a separate thread from the main thread that is used to send the test requests. The test application sends a request to the API every 100 millisecond for the total of 1000 requests. The direct OPC UA request test is done by a python script. The test script sends a service request to the OPC UA server every 50 milliseconds for the total of 1000 requests. Lower request interval for the OPC UA direct testing is justified by the less time it takes to complete a single request.

Figure 15 shows results for querying the GraphQL API which in turn requests the OPC UA server for the results. The tests were run on the control application with a script made specifically for testing. Requests were sent every 100 milliseconds for the total of 1000 requests. The request completion times consistently fall to 11-13 millisecond range with a few instances of 14 millisecond completion times. The average request completion time was 11,5 milliseconds.

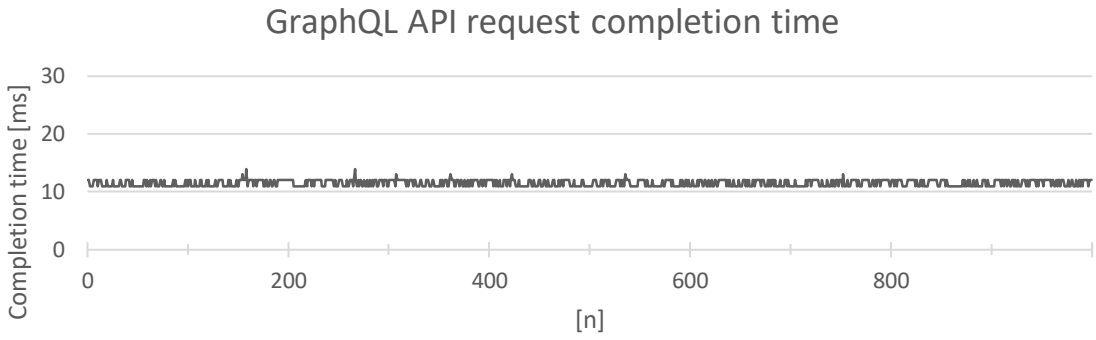

Figure 15. Request completion time in a controlled environment. 100 millisecond intervals between requests.

Figure 16 shows results for requesting the OPC UA server directly. The test service requests are done by a Python script. The script batches the two value changes to a single request, just like querying the GraphQL API does. The service requests are sent every 50 milliseconds for total of 1000 service requests. Request completion times fall to range of 0,94-1,61 milliseconds. On average the requests took 1,27 milliseconds to complete which is considerably faster than the average of 11,5 milliseconds when querying the GraphQL API. From this it can be said that the GraphQL API adds up around 10 millisecond latency compared to directly requesting the OPC UA server.

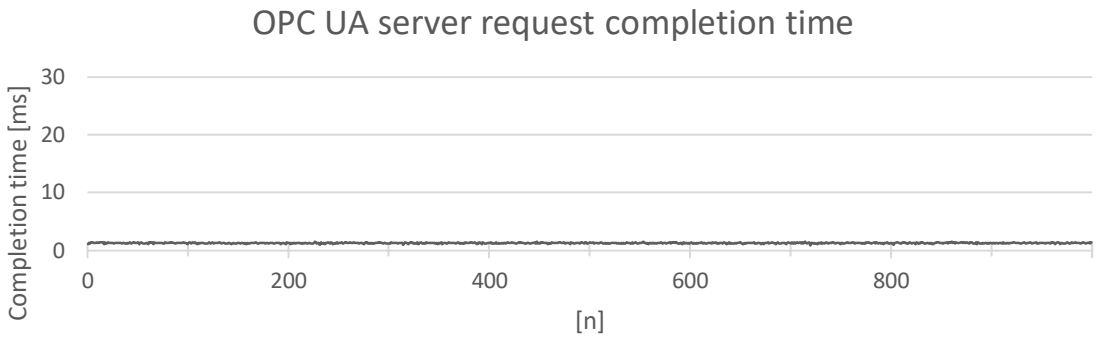

Figure 16. Request completion time in a controlled environment directly to OPC UA server. 50 millisecond intervals between requests.

Figure 17 has data on the requests when the testing control application has watchdog running in the background in addition to the test requests every 100 milliseconds. The watchdog sends a similar request as the test requests to the GraphQL API every 250 milliseconds. The watchdog is run on a separate thread from the main test request thread. Running the watchdog in the background seems to have caused spikes in request completion time periodically. Curiously, the spikes seen in Figure 17 seem to occur approximately every 4,5 seconds once taking into account that, each request (n) is done in 100 millisecond intervals and each watchdog request in 250 millisecond intervals. Other than the spikes, the base request completion time is around the same 11-13 millisecond range as seen in Figure 15. Overall, the requests fall between 10-24 milliseconds. Average request completion time was 12,14 milliseconds.

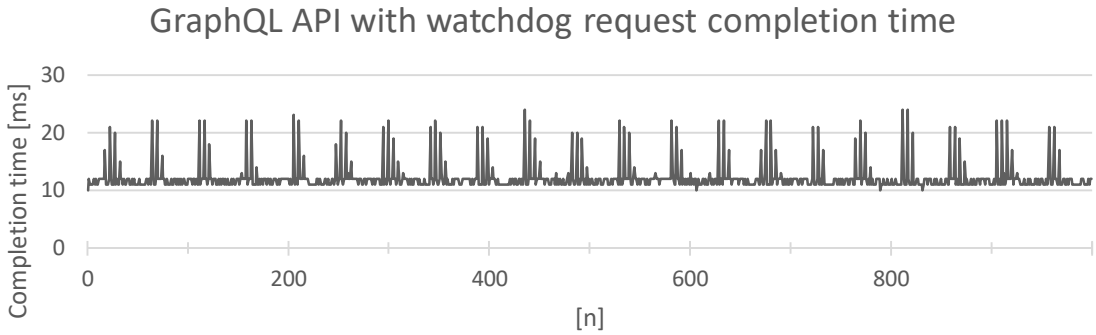

Figure 17. Request completion time in a controlled environment. 100 millisecond intervals between requests with watchdog requesting every 250 milliseconds.

3.2.2 Crane control use case

Crane control use case tests are done with the control application in a similar situation as when actually controlling the crane. However, the tests are done with the same testing script as the tests in controlled environment. The scripts simulate a situation where control signals are sent to the crane with GraphQL API requests. Directly requesting the OPC UA server is also tested, but the test software used is a python script instead of the control application, and no watchdog is running in the background. Thus, the direct OPC UA tests are done only to give context to the efficiency of the requests sent to the GraphQL API. Just like in the controlled environment tests, the requests are sent every 100 or 50 milliseconds depending

on if the GraphQL API or OPC UA server is being tested. Requests are sent for the total of 1000 to detect any possible inconsistencies in the request completion times. Increasing the request interval to longer than 100 milliseconds has been observed to not affect the results in a noticeable way.

The tests are done with both a mobile device and a PC. Mobile device tests are done both close to the crane and far away from the crane with relatively bad Wi-Fi connection. The close to the crane tests are done from around 10 meters to the router that the GraphQL API server is connected to. The faraway (i.e. bad connection) tests include a Wi-Fi repeater in the middle that routes the requests to the Wi-Fi router that is connected to the GraphQL with an ethernet cable. The Wi-Fi repeater is approximately 20 meters from the crane with line of sight, and approximately 30 meters away from test client with two walls in between. Some context to the test setup topology can be seen in Figure 13. The client in the figure is the same as the test client in these test cases and the Wi-Fi repeater is the same as which is used in the weak connection tests. The Wi-Fi connection used in tests is of 2,4 GHz frequency, but a single test is done with 5 GHz connection for comparison.

Figure 18 has data on request completion time when a mobile client is approximately 10 meters from the crane. The conditions are as if user was controlling the crane with a mobile phone, just like in real use. Some large time spikes up to 400 milliseconds can be seen which are likely caused by momentary traffic loads or packet losses between the client and the GraphQL API. Other than the spikes, the request completion times range from 58 to 178 milliseconds. The average completion time is at 90 milliseconds.

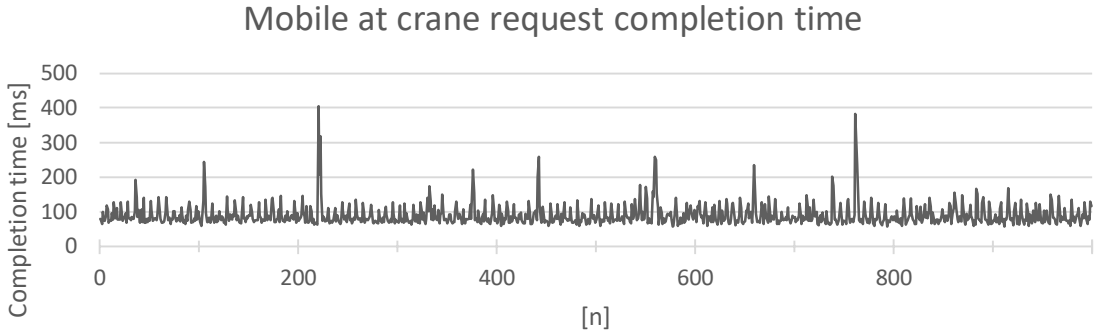

Figure 18. Request completion time in a real use case close to the crane. 100 millisecond intervals between requests with watchdog requesting every 250 milliseconds.

Figure 19 has data on request completion time in an almost similar test setup as in Figure 18. Only exception is that the client is further away from the crane. More definitely, the data is directed through a Wi-Fi extender from which the mobile client is approximately 30 meters away. Additionally, there are two walls in between the Wi-Fi extender and the client. This causes noticeably more and higher spikes in the request completion times, some rising up to the 500-millisecond limit at which the request is timed out. Thus, request completion times range from 63 to 501 milliseconds. The average completion time is at 111 milliseconds.

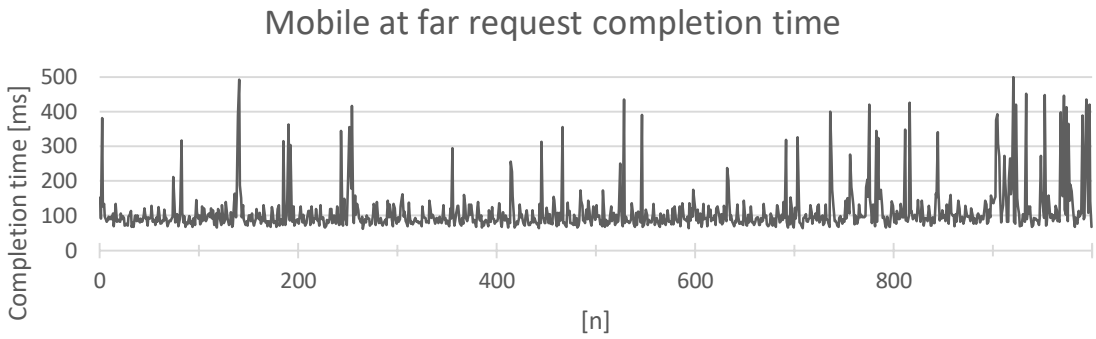

Figure 19. Request completion time in a real use case far away from the crane. 100 millisecond intervals between requests with watchdog requesting every 250 milliseconds.

Figure 20 also has data from a faraway test. This time the difference is that instead of making the requests with a mobile phone, a PC is used. A PC has potentially stronger wireless networking capabilities which might affect the completion times. From the data can be seen that the spikes are now more alike in Figure 18 and the variations in request completion times are smaller. The reasons for this can range from better Wi-Fi antennas to better computational performance compared to the mobile device. Request completion times range from 52 to 366 milliseconds. The average completion time is at 79 milliseconds. Interestingly, based on the average, the PC seemed to perform better than when the mobile device is close to the crane.

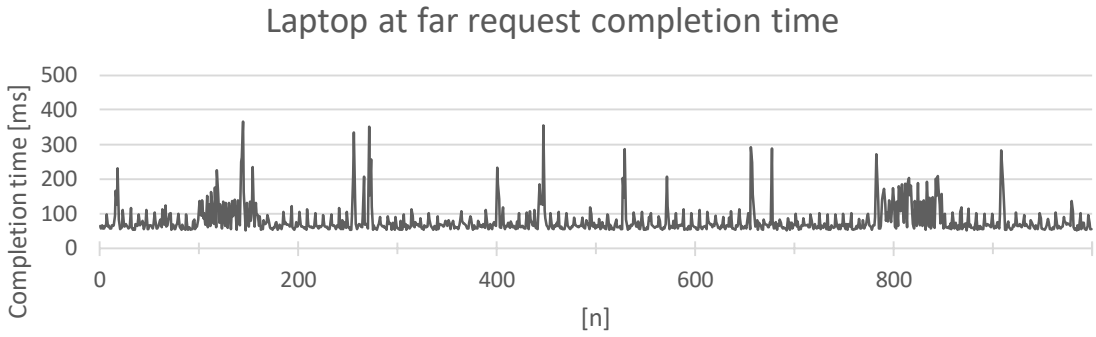

Figure 20. Request completion time in a real use case with a PC far away from the crane. 100 millisecond intervals between requests with watchdog requesting every 250 milliseconds

Figure 21 also has data collected similarly with a PC and far away from the crane. However, the Wi-Fi connection frequency used was changed from standard 2,4 GHz to 5GHz. The hypothesis of the different frequency is that 5 GHz Wi-Fi network is commonly less crowded to its lower frequency counterpart leading to less interference to the connection. In addition, the 5 GHz has noticeably faster data transfer speeds. These points might have an effect to the potential packet drops that can occur with the 2,4 GHz Wi-Fi and overall speed performance. One downside to the 5GHz connection is that due to higher frequency the signal travels a bit worse through solid walls and obstacles. The Figure 21 data does in fact show that the request completion times are almost consistently below 100 milliseconds. The spikes as seen in other figures are not present here either. An instance of subsequent higher completion times can be seen in the figure. This could be caused by some other traffic in the

network during the test because it does not seem to be random. The request completion times range from 51 to 286 milliseconds with an average of 70 milliseconds. This means that this test setup performed the best out of all real use case tests via the GraphQL API.

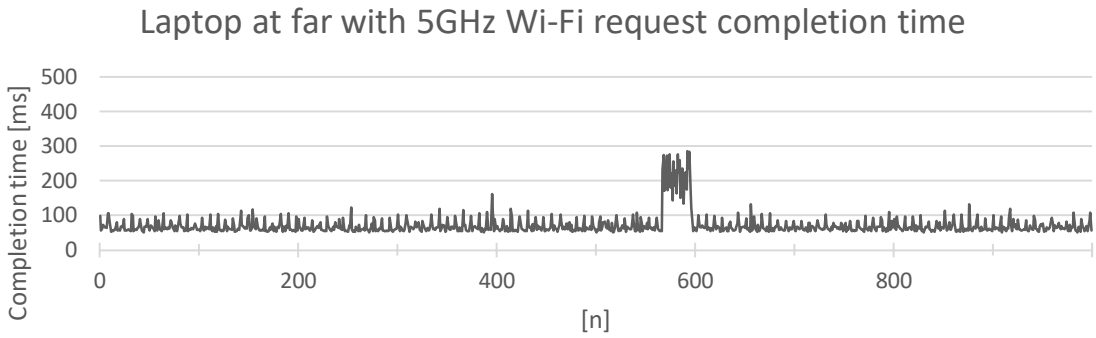

Figure 21. Request completion time in a real use case with a PC far away from the crane. Wi-Fi 5 GHz instead of 2,4 GHz. 100 millisecond intervals between requests with watchdog requesting every 250 milliseconds

Finally, the Figure 22 data is collected by using the same PC as before, but sending the requests directly to the OPC UA server using a Python script. Differing from rest of the use case tests, no watchdog was running in the background. While the average is significantly lower than in other tests, some spikes can still be seen in the data. This could once again point to the possibility of Wi-Fi connection being unreliable for consistent data transfer. Overall, lower request completion times could be caused by smaller packet sizes when using the OPC UA communication protocol instead of HTTP requests. The request completion times range from 5 to 306 milliseconds, with average of 12,5 milliseconds.

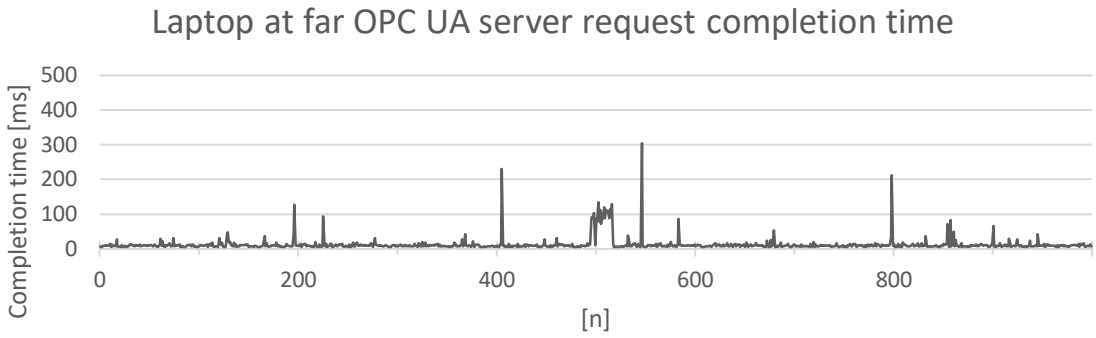

Figure 22. Request completion time in crane control environment with a PC far away from the crane. Requesting OPC UA server directly. 50 millisecond intervals between requests.

Table 8 has collected general information from each request completion time figure's data. Each datasets average, minimum and maximum, and standard deviance can be found from the table. The average value help determine in each test case how long it takes for a request return a response. Minimum and maximum values give the range to which the request completion time will most likely fall in between. The standard deviation gives a hint on the stability of the completion times from request to request. Larger standard deviation can be seen in the test cases corresponding figure as random spikes and noise. As expected, the

standard deviation is smallest in test cases where the connection is considered better. The better connection can be caused by being closer to the crane or using better hardware i.e. PC instead of a mobile device. In controlled environment tests, the standard deviation is considerably smaller due to the connection between servers being fast and stable.

Table 8. Request completion times with average, minimum, maximum values, and standard deviation.

Environment	*Test target/device*	*Avg*	*Min*	*Max*	*σ*
Controlled environment	GraphQL API	11,5	11	14	0,5
	OPC UA server	1,3	1,6	0,9	0,1
	GraphQL API with watchdog	12,1	10	24	2,3
Real use case	Mobile at crane	89,9	58	406	31,5
	Mobile at far	110,9	63	501	63,1
	Laptop at far	79,0	52	366	39,1
	Laptop at far 5GHz Wi-Fi	70,3	51	286	29,7
	Laptop at far OPC UA direct	12,5	5	306	20,2

When the connection is optimal, as is in the controlled environment tests, from the results can be said that requesting through the GraphQL API adds around 10,2 milliseconds to the request completion time. In real use case, the GraphQL API adds up around 68 milliseconds to the request completion time. This difference in speed between the controlled and uncontrolled environment when requesting the GraphQL API is quite large compared to directly requesting the OPC UA server. It seems likely that the HTTP request protocol uses larger packets that travel longer over the network compared to the OPC UA service requests.

The communication protocol packet sizes are different when using the GraphQL API or OPC UA server directly. With Wireshark application, communication packet sizes and traffic can be measured (Wireshark Foundation, 2010). The data collected from the Wireshark can be seen in Table 9. The results show that requests to the GraphQL API send considerably more data than when requesting the OPC UA server directly. This can be one factor in higher average request completion times when requesting the GraphQL API. In addition to the larger request size, the GraphQL API server must also request the OPC UA server after receiving the query, which should add up some latency. This OPC UA service request sent by the Graph QL API is similar to the test request sent directly to the OPC UA server but uses an ethernet connection instead of the Wi-Fi.

Table 9. Data transferred over the network during test requests.

	Request [bytes]	*Response [bytes]*	*Total [bytes]*
GraphQL API	749	98	847
OPC UA direct	242	132	374

4 Discussion and Conclusion

In this section, the developed GraphQL API's success in terms of goals achieved is discussed. First the design choices made when developing the GraphQL API are discussed on whether the requirements were fulfilled or not. Also, the API is compared to other almost similar solutions developed by other researchers. Next, the performance results are analysed as to whether the GraphQL API is fast enough for its purpose. Finally, the thesis is concluded by final words on the project and its success.

4.1 Discussion

At the start of this thesis project, the question of *how can we simplify the OPC UA server communication to be friendlier with common developers* was asked. The development of the GraphQL API was guided by the motivation of making things easier for developers. With the GraphQL API the developer does not need to know how to communicate with an OPC UA server in order to consume its data. The developer only needs to know how to fetch data from a GraphQL API. GraphQL has the benefit of being well documented and more familiar to developers when comparing to OPC UA. GraphQL APIs are also considered difficult to "get wrong" when developing as the architecture is well defined in the documentation. Similar to OPC UA Information Model, GraphQL views the resources and their relations in an object-like manner. The capabilities were also similar in that only the resources that the client asked for are normally fetched. Some compromises, however, had to be made with supported OPC UA features to ensure that the GraphQL API was logical and easy to understand to the users. Time constraints also had an effect on what features were feasible to implement to the GraphQL API.

Another question to which an answer was searched for was *how much latency can the resulted API cause while still being usable in most real-time and user applications*. The GraphQL API was built to add minimal latency to the completion of requests. The resulted API should be capable of handling most real-time applications just as an OPC UA server would. However, as the results showed, the requests to the API took significantly longer, which could be attributed at least in part to the HTTP protocols high latency and overhead (Ala-Laurinaho, 2019). But, as is discussed in the following Performance section, the latencies are still at acceptable level for most applications.

The web-based control application built as a case study deserves a mention as it has potential for impressive applications. The mobile devices it can be run on commonly have multiple sensors which can be accessed in the JavaScript. This means that sensors such as accelerometer, GPS and touchscreen can be augmented in the Ilmatar crane's control with the control application. In addition, the control application could potentially even share its location or other information with other users accessing Ilmatar's user interface. For example, the crane could be prevented from moving over other user's location as a safety measure. The control application can combine data in its monitor and control logics from the crane's GraphQL API, other local resources, device's internal sensors, and even any online web resource if the application has access to public internet. Basically, from any resource that is accessible from the JavaScript running the application.

4.1.1 Solution

The requirements, as specified in GraphQL API Requirements section, were met in the final solution of the GraphQL API. Reading and writing values to OPC UA nodes via the

GraphQL API is possible. Required OPC UA server information model elements, such as nodes and attributes were successfully exposed via the GraphQL API. Similar to the OPC UA service requests, it is possible to read or write multiple values with a single query. In addition to the GraphQL API solution fulfilling the requirements, some nice-to-have features were also added. These nice-to-have functions have options to explore the OPC UA server address space with a graphical user interface, configure new OPC UA servers for the GraphQL API to aggregate, and view the configured OPC UA servers' settings. None of these operations require the user to access the backend. Instead, these operations can be done using the same GraphQL interface that is used to read and write values to the OPC UA server. Overall, the GraphQL API turned out to be a functional web interface for any OPC UA specification compliant OPC UA server.

The GraphQL API was developed as a solution to easier access to OPC UA server data. There were multiple ways to complete the requirements. Thus, the solution presented in this thesis is just one of the many possible ways to solve the problem of OPC UA server data accessibility. Other companies and researchers have built other kinds of APIs, most of which focus on the RESTful style of API. In some important features of different API solutions for OPC UA servers are listed. For comparison two other web API solutions for OPC UA servers are listed with the GraphQL API. Grüner *et al.* (2016) developed a RESTful API integrated as part of the OPC UA server (Grüner, Pfrommer and Palm, 2016). The RESTful OPC UA was designed to receive only requests that were session independent on the OPC UA server. Meaning, basic read and write services are possible, but subscriptions were not supported. Also, due to the nature of the RESTful API being integrated to the OPC UA server, it does not aggregate multiple OPC UA servers under the same API. Cavalieri *et al.* (2019) instead, have built a RESTful OPC UA Web Platform which is an independent entity of the OPC UA server, much like the GraphQL API (Cavalieri, Salafia and Scroppo, 2019). Thus, it is also capable of aggregating multiple different OPC UA servers. OPC UA Web Platform similarly to GraphQL API communicates with the OPC UA server as client. Operations that the platform supports are read, write, and subscription of value changes. However, an option for adding new Nodes to the OPC UA servers was not supported. The GraphQL API mainly differs from the OPC UA Web Platform because it uses GraphQL instead of RESTful, and GraphQL API has an option to manage Nodes on the OPC UA servers. Table 10 has a summarized comparison on the presence of important OPC UA features between the RESTful OPC UA, OPC UA Web Platform and GraphQL API.

Table 10. Comparison of the GraphQL API to other existing HTTP-based APIs for OPC UA in literature. [1](Grüner, Pfrommer and Palm, 2016), [2](Cavalieri, Salafia and Scroppo, 2019)

	Read	***Write***	***Subscription***	***Aggregate***	***Manage Nodes***
RESTful OPC UA[1]	Yes	Yes	No	No	Yes
OPC UA Web Platform[2]	Yes	Yes	Yes	Yes	No
GraphQL API	Yes	Yes	No	Yes	Yes

The built GraphQL API does not support subscriptions at the time of writing this thesis, due to required libraries still being in the middle of development. For the subscriptions to work in the current build of the GraphQL API, it would be necessary to rollback some core

libraries to older releases. This would cause problems with other core functionalities that were built on newer code. However, in the near future subscription support should be possible to be added to the existing GraphQL API with little additional coding required. User authentication is neither supported at the time of writing, but can also be added if necessary. Authentication support was not added to the GraphQL API because the Ilmatar OPC UA server does not have authentication requirements when reading or writing information to relevant nodes. GraphQL was chosen as the API technology for this project in part due to GraphQL being designed to have these features available. The decision to create the application in Python with Django and Graphene-Django -GraphQL library has, however, delayed the addition of the subscription feature because of the immature libraries.

GraphQL API server also has the benefit of being easily customisable. Adding new fields to the types is as simple as writing the additional field to a type and a resolver function for the field. In theory, it is possible to expose the whole OPC UA information model via the GraphQL API similarly to as it is presented on the OPC UA server. GraphQL type structure is quite similar to the OPC UA information model, as both are object structured. Presenting the whole OPC UA information model via the GraphQL API was, however, not deemed necessary. Most of the attributes and nodes on the OPC UA server are not required in the context of reading sensor values and controlling the Ilmatar crane. Thus, they were not listed in the GraphQL API requirements either. Another convenient customisation feature with GraphQL is the possibility to deprecate outdated fields. For example, if in the future the GraphQL API is modified and some fields have become outdated, it is possible to mark the fields as deprecated. Deprecated fields are hidden for future applications, but the field is still available for older applications that have not been updated yet. (GraphQL Foundation, 2019). Field deprecation ensures that the GraphQL API is compatible with older applications even if the development has greatly modified the resources being served.

Lastly, GraphQL queries are customisable on the client side so that unrequested fields are not retrieved from the OPC UA server. This results in somewhat lower latency due to less information to fetch from the OPC UA server, and the HTTP request and response payloads are slightly smaller. If the application requires low latency, it is recommended to only fetch what is necessary from the GraphQL API. This also helps with the overall load that might hit the GraphQL API and OPC UA servers.

4.1.2 Performance

The performance (i.e. request completion time) is measured to give an estimation if the crane is possible to control and monitor simultaneously via the GraphQL API. In addition, the measurements also gave context on the added latency compared to using the OPC UA server directly. As was one of the requirements in this work, the latency from when inputting a control signal to the crane starting movement should be low enough. The crane controls should not have noticeable input latency. However, the crane is a heavy object that takes a while to start noticeably moving after a movement command. Thus, users do not expect the crane to respond to control signals instantly. A good comparison could be made by comparing the control application latency to the radio controller's latency. But, this kind of test is difficult to arrange due to not having access to the software on the radio controller. Other inaccuracies could be caused by the dead-zone in the radio controller joysticks and the difficulty of measuring the crane movement in relation to physical inputs. These factors lead to the results comparing only the request completion times.

The results received for the system were roughly around 70 milliseconds to 110 milliseconds in average. This result is a measurement of the round-trip time that the request took to receive a response. The actual latency to the crane receiving the control signal should be smaller because the response is not necessary to be received before the value on the OPC UA server has changed. To give context to the latencies, it can be compared to remote control situations where a low latency is crucial. Research has been made by Anvari *et al.* (2005) on how high of a latency affects the success of a remote surgery (Anvari *et al.*, 2005). The study concluded that some operations have been successfully completed at the latency of 135 milliseconds. In addition to this, Marescaux *et al.* (2016) has in their research carried out an remote surgery both successfully and safely with a latency of 155 milliseconds (Marescaux *et al.*, 2006). Comparing these latencies to the average of 110 milliseconds worst case scenario when controlling the crane, it can be concluded that the latency should be low enough. The crane does not necessarily require as precise control either, compared to a surgery, and generally users do not expect the crane to response to controls as precisely as surgical equipment. In better control environments, with less Wi-Fi interference, the latencies can drop to around 70 milliseconds and below.

The overcrowded 2,4 GHz Wi-Fi frequency that was used for most of the real-use tests caused random spikes to the request completion times. These spikes also affected the average completion time. Changing the Wi-Fi frequency to 5 GHz noticeably dropped the latency down to 70 milliseconds from 80 milliseconds, in an otherwise similar test case, and removed the random spikes completely. One reason for this could be that the newer 5 GHz Wi-Fi is not as crowded as the older 2,4 GHz Wi-Fi. This indicates that it is desirable for best performance to ensure that the Wi-Fi connection is not receiving interference from other Wi-Fi networks in the area. Using the 5 GHz network is also possible, but the Wi-Fi range would not be as good as with the 2,4 GHz connection. In the tests, however, the 5 GHz network was equally usable even in the bad connection test case.

Another effect to the performance was caused by the device running the test control application. Running the control application on a PC instead of a mobile device had a noticeable effect on the request completion times. The average latency dropped from 110 to 80 milliseconds. This can partly be explained by better computational performance and possibly even by the Wi-Fi adapter being more capable on the PC compared to the mobile device. Additional reasons can be that the mobile browser handles constant requesting differently than the PC version of the browser.

When testing in a controlled environment, the GraphQL API and OPC UA direct latencies were both steady and had a rough average difference of 10 milliseconds. This implies that the GraphQL API itself adds 10 milliseconds due to the way it handles incoming queries and forwarding the request to the OPC UA server. Most of the latency is caused by having to send multiple service requests to the OPC UA server. These service requests add up total time even when all the connections are internal to the test PC.

For lower latencies, some more development and additional upgrades can be made to the system. The hardware used can be upgraded to faster servers and network devices. The Wi-Fi routers and servers were mostly used because they were either readily available, or good platforms to develop for. The GraphQL API software also has some room for optimization. Some unnecessary checks are still made to the OPC UA server by the GraphQL API which can be optimized in the code. Additionally, the subscription feature is possible to implement

once the required libraries have been developed a bit further. The subscription feature would streamline the communication between the client, the GraphQL API and the OPC UA server.

4.2 Conclusion

The target was to build a web interface that enhances the communication between the Ilmatar crane and its digital twin components. The developed GraphQL API managed to support most relevant features of the OPC UA server that were required from the Ilmatar crane. An upside compared to directly using the OPC UA server is that the user is not required to study the OPC UA specification to start developing with the GraphQL API. Additionally, GraphQL created with developers in mind which can be seen in its growing popularity among web services. For Ilmatar crane, GraphQL API simplifies the addition of new components to the Ilmatar's digital twin. The control software developed as a case study is one proof of this. The GraphQL API and the control application has already helped in building a new control functionality for the Ilmatar crane by a third-party developer. The built component was a location tracking feature which was included as part of the control application. The developer did not require any knowledge of the OPC UA server information model when designing the control functionality. The tracking software successfully made the Ilmatar crane follow a portable beacon in real-time. The beacon's position was tracked by its own set sensors and the value matched to the location data on the Ilmatar. This required continuous data exchange between the control application, OPC UA server via the GraphQL API, and the beacon's position tracking software. Thus, the GraphQL API proved to performant enough even for some automated control applications.

The GraphQL API is easy to access in code from virtually any device with only basic HTTP request library support required. This ensures the access to the Ilmatar crane information even from environments where the OPC UA library support is not available. The only downside to the GraphQL API, compared to directly using the OPC UA server, is the latency which ended up being around five times higher in use case scenarios. However, even the higher latency is considered to be fast enough for real-time control and communication. In contrast, remote medical operations are successfully completed at latencies higher than the latencies experienced with the GraphQL API. Either way, some applications may still require even lower latencies or energy efficiency for which the GraphQL API, or any web API, might not be good enough. The GraphQL API is possible to be further developed for more optimal performance. The hardware could be upgraded to faster servers as the hardware used was mostly what was considered good enough. The GraphQL API can also be developed a bit further for more optimized communication with the OPC UA server. Also, the requested subscription feature did not make it to the finished software but it possible to be implemented in the future.

The GraphQL API was developed foremost for Ilmatar's digital twin. However, it can also be used to enhance any digital twin that uses an OPC UA server as the data link between the physical object and its digital twin. A lot of digital twin's features can now be developed fully web based and most of the data can be easily fetched via the GraphQL API. This work potentially brings digital twin development to a broader developer base which can in turn accelerate the development of digital twins as a whole. To aid the development of digital twins, the GraphQL API software has been released as open source on GitHub under the Aalto Industrial Internet Campus organization (Hietala, 2019). Along with other digital twin related projects, it is available for further development efforts by interested developers.

References

Ala-Laurinaho, R. (2019) Sensor Data Transmission from a Physical Twin to a Digital Twin. Aalto University. Available at: http://urn.fi/URN:NBN:fi:aalto-201905123028.

Anvari, M. *et al.* (2005) The impact of latency on surgical precision and task completion during robotic-assisted remote telepresence surgery, *Computer Aided Surgery*, 10(2), pp. 93–99. doi: 10.1080/10929080500228654.

Autiosalo, J. *et al.* (2019) A feature-based framework for structuring industrial digital twins, *IEEE Access*, pp. 1–1. doi: 10.1109/ACCESS.2019.2950507.

Bielefeldt, B., Hochhalter, J. and Hartl, D. (2015) Computationally efficient analysis of SMA sensory particles embedded in complex aerostructures using a substructure approach, in *ASME 2015 Conference on Smart Materials, Adaptive Structures and Intelligent Systems, SMASIS 2015*. ASME, p. V001T02A007. doi: 10.1115/SMASIS2015-8975.

Byron, L. and Contributors (2015) DataLoader, *github.com/graphql/dataloader*. Available at: https://github.com/graphql/dataloader (Accessed: 28 August 2019).

Cavalieri, S., Salafia, M. G. and Scroppo, M. S. (2019) Integrating OPC UA with web technologies to enhance interoperability, *Computer Standards and Interfaces*. Elsevier B.V., 61, pp. 45–64. doi: 10.1016/j.csi.2018.04.004.

Chesneau, B. *et al.* (2016) Gunicorn - Python WSGI HTTP Server for UNIX, *gunicorn.org*. Available at: http://gunicorn.org/ (Accessed: 28 August 2019).

Django Software Foundation (2018) Django - The Web framework for perfectionists with deadlines*, Django Software Foundation*. Available at: https://www.djangoproject.com/ (Accessed: 28 August 2019).

Facebook Inc. (2016) GraphQL - A query language for your API, *GraphQL Homepage*. Available at: https://graphql.org/ (Accessed: 20 August 2019).

Fielding, R. *et al.* (1999) Hypertext Transfer Protocol -- HTTP/1.1, *The Internet Society*. Available at: http://www.hjp.at/doc/rfc/rfc2616.html (Accessed: 17 November 2019).

FreeOpcUa (2018) FreeOpcUa: Open Source C++ and Python OPC-UA Server and Client Libraries and Tools. Available at: http://freeopcua.github.io/ (Accessed: 6 August 2019).

Glaessgen, E. H. and Stargel, D. S. (2012) The digital twin paradigm for future NASA and U.S. Air force vehicles, *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, pp. 1–14. doi: 10.2514/6.2012-1818.

GraphQL Foundation (2018) Who's Using, *graphql.org*. Available at: https://graphql.org/users/ (Accessed: 13 August 2019).

GraphQL Foundation (2019) GraphQL Specifications, *graphql.github.io*. Available at: https://graphql.github.io/graphql-spec/draft/ (Accessed: 15 August 2019).

Grieves, M. and Vickers, J. (2016) Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems, in *Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches*. Cham: Springer International Publishing, pp. 85–113. doi: 10.1007/978-3-319-38756-7_4.

Grüner, S., Pfrommer, J. and Palm, F. (2016) RESTful Industrial Communication with OPC UA, *IEEE Transactions on Industrial Informatics*. IEEE Computer Society, 12(5), pp. 1832–1841. doi: 10.1109/TII.2016.2530404.

Harmo, P. (2014) OPC ja OPC UA, Novotek. Available at: https://www.novotek.com/fi/ratkaisut/kepware-kommunikointialusta/opc-ja-opc-ua (Accessed: 18 July 2019).

Haro, C. *et al.* (2016) GraphQL Python, *github.com/graphql-python*. Available at: https://github.com/graphql-python (Accessed: 28 August 2019).

Hietala, J. (2019) GraphQL API for OPC UA servers, *github.com/AaltoIIC*. Available at: https://github.com/AaltoIIC/OPC-UA-GraphQL-Wrapper (Accessed: 20 November 2019).

Kurose, J. F. and Ross, K. W. (2013) Computer Networking: A Top-Down Approach: 6th edition. Pearson Education UK. ISBN 9780273775638.

Laaki, H., Miche, Y. and Tammi, K. (2019) Prototyping a Digital Twin for Real Time Remote Control over Mobile Networks: Application of Remote Surgery, *IEEE Access*. Institute of Electrical and Electronics Engineers Inc., 7, pp. 20235–20336. doi: 10.1109/ACCESS.2019.2897018.

Lämmer, L. and Theiss, M. (2015) Product lifecycle management, *Concurrent Engineering in the 21st Century: Foundations, Developments and Challenges*, pp. 455–490. doi: 10.1007/978-3-319-13776-6_16.

Leitner, S.-H. and Mahnke, W. (2006) OPC UA – Service-oriented Architecture for Industrial Applications, *ABB Corporate Research Center*, 26(4), pp. 1–6. Available at: http://cimug.ucaiug.org/kb/knowledge base/soa for industrial applications.pdf.

Marescaux, J. *et al.* (2006) Transcontinental robot-assisted remote telesurgery, feasibility and potential applications, *Teleophthalmology*. doi: 10.1007/3-540-33714-8_31.

Matrikon (2019) OPC Data eXchange (OPC DX) 1.00 Specification. Available at: https://www.matrikonopc.com/downloads/143/specifications/index.aspx (Accessed: 6 August 2019).

MDN Web Docs (2019a) HTTP Headers, *developer.mozilla.org*. Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers (Accessed: 17 November 2019).

MDN Web Docs (2019b) HTTP Messages, *developer.mozilla.org*. Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages (Accessed: 17 November 2019).

Mukherjee, T. and DebRoy, T. (2019) A digital twin for rapid qualification of 3D printed metallic components, *Applied Materials Today*. Elsevier Ltd, 14, pp. 59–65. doi: 10.1016/j.apmt.2018.11.003.

NGINX (2017) NGINX Wiki, *nginx.com*. Available at: https://www.nginx.com/resources/wiki/ (Accessed: 28 August 2019).

OPC Foundation (2015) What is OPC? Available at: https://opcfoundation.org/about/what-is-opc/ (Accessed: 26 June 2019).

OPC Foundation (2017a) OPC UA Part 1: Overview and Concepts. OPC Foundation. Available at: https://opcfoundation.org/developer-tools/specifications-unified-architecture.

OPC Foundation (2017b) OPC UA Part 3: Address Space Model Specification. OPC Foundation. Available at: https://opcfoundation.org/developer-tools/specifications-unified-architecture.

OPC Foundation (2017c) OPC UA Part 4: Services. OPC Foundation. Available at: https://opcfoundation.org/developer-tools/specifications-unified-architecture.

OPC Foundation (2017d) OPC UA Part 6: Mappings. OPC Foundation. Available at: https://opcfoundation.org/developer-tools/specifications-unified-architecture.

Pallets (2010) Flask, *flask.palletsprojects.com*. Available at: https://flask.palletsprojects.com/en/1.0.x/ (Accessed: 8 October 2019).

Raspberry Pi Foundation (2019) Raspberry Pi hardware - Raspberry Pi Documentation, *raspberrypi.org*. Available at: https://www.raspberrypi.org/documentation/hardware/raspberrypi/ (Accessed: 28 August 2019).

Raspbian.org (2016) Raspbian, *raspbian.org*. Available at: https://www.raspbian.org/ (Accessed: 28 August 2019).

Rudin, D. (2011) OWFa 1.0, *Open Web Foundation*. Available at: http://www.openwebfoundation.org/legal/the-owf-1-0-agreements/owfa-1-0 (Accessed: 13 August 2019).

Rykovanov, A. and Oroulet (2014) Free OPC-UA Library, *github.com/FreeOpcUa*. Available at: https://github.com/FreeOpcUa (Accessed: 26 August 2019).

Sanner, M. F. (1999) Python: A programming language for software integration and development, *Journal of Molecular Graphics and Modelling*. Available at: https://pdfs.semanticscholar.org/409d/3f740518eafcfaadb054d9239009f3f34600.pdf (Accessed: 28 August 2019).

Söderberg, R. *et al.* (2017) Toward a Digital Twin for real-time geometry assurance in individualized production, *CIRP Annals - Manufacturing Technology*. Elsevier USA, 66(1), pp. 137–140. doi: 10.1016/j.cirp.2017.04.038.

Stubailo, S. (2017) GraphQL vs. REST – Apollo GraphQL, *blog.apollographql.com*. Available at: https://dev-blog.apollodata.com/graphql-vs-rest-5d425123e34b (Accessed: 8 October 2019).

Tao, F. *et al.* (2018) Digital twin-driven product design, manufacturing and service with big data, *International Journal of Advanced Manufacturing Technology*. The International Journal of Advanced Manufacturing Technology, 94(9–12), pp. 3563–3576. doi: 10.1007/s00170-017-0233-1.

Tao, F. and Zhang, M. (2017) Digital Twin Shop-Floor: A New Shop-Floor Paradigm Towards Smart Manufacturing, *IEEE Access*, 5, pp. 20418–20427. doi: 10.1109/ACCESS.2017.2756069.

Torikian, G. *et al.* (2016) The GitHub GraphQL API, *github.blog*. Available at: https://github.blog/2016-09-14-the-github-graphql-api/ (Accessed: 25 June 2019).

Unified Automation GmbH (2019) High Performance OPC UA Server SDK: OPC UA NodeId Concepts, *Documentation.Unified-Automation.Com*. Available at: https://documentation.unified-automation.com/uasdkhp/1.3.0/html/_l2_ua_node_ids.html (Accessed: 25 June 2019).

W3Schools (2016) JavaScript Versions, *w3schools.com*. Available at: https:// www.w3schools.com/js/js_versions.asp (Accessed: 4 January 2020).

Wireshark Foundation (2010) Wireshark, *wireshark.org*. Available at: http:// www.wireshark.org/ (Accessed: 8 November 2019).